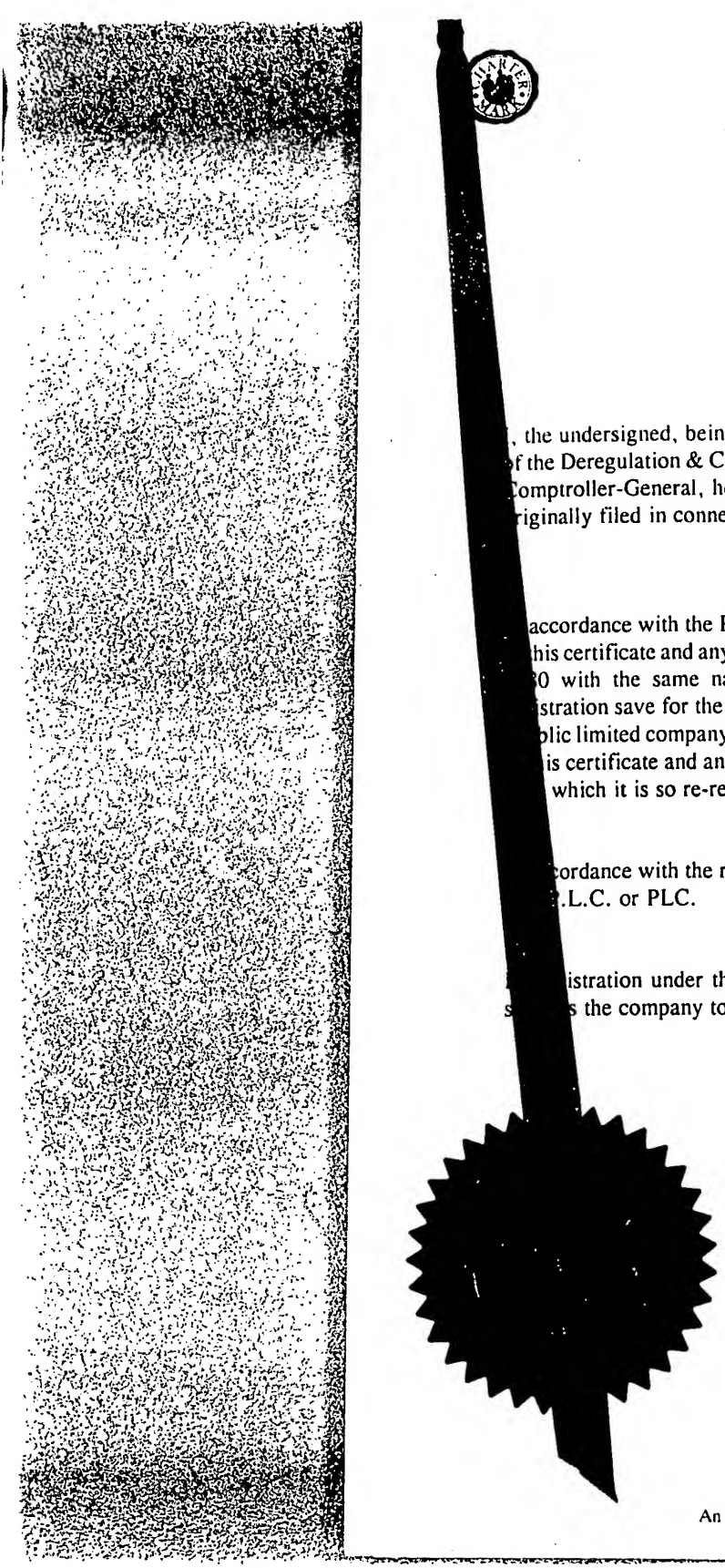


The Patent Office

INVESTOR IN PEOPLE

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) f the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as riginally filed in connection with the patent application identified therein.

accordance with the Patents (Companies Re-registration) Rules 1982, if a company named his certificate and any accompanying documents has re-registered under the Companies Act 0 with the same name as that with which it was registered immediately before re-istration save for the substitution as, or inclusion as, the last part of the name of the words blic limited company" or their equivalents in Welsh, references to the name of the company is certificate and any accompanying documents shall be treated as references to the name which it is so re-registered.

ordance with the rules, the words "public limited company" may be replaced by p.l.c., .L.C. or PLC.

istration under the Companies Act does not constitute a new legal entity but merely s the company to certain additional company law rules.

Signed

Dated 17 March 2004

CERTIFIED COPY OF PRIORITY DOCUMENT

Patents Form 1/77

The Patent Office
Cardiff Road
Newport
NP9 1RH

Request for grant of a patent

1.	Your Reference	**IMR/CEE/Y1402** 16JUL03 E822773-2 D02846 P01/7700 0.00-0316532.1
2.	Application number	15 JUL 2003 **0316532.1**
3.	Full name, address and postcode of the or each Applicant Country/state of incorporation (if applicable)	**Transitive Limited** **5th Floor Alder Castle** **10 Noble Street** **London** **EC2V 7QJ** **Incorporated in: United Kingdom**
	08664211001	
4.	Title of the invention	**Method and Apparatus for Partitioning Code in Program Code Conversion**
5.	Name of agent	**APPLEYARD LEES**
	Address for service in the UK to which all correspondence should be sent	**15 CLARE ROAD** **HALIFAX** **HX1 2HY**
	Patents ADP number	**190001** ✓

6. Priority claimed to:

Country	Application number	Date of filing

7. Divisional status claimed from:

Number of parent application	Date of filing
-	-

8. Is a statement of inventorship and of right to grant a patent required in support of this application? **YES**

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document

Continuation sheets of this form	
Description	**84**
Claim(s)	**17**
Abstract	**1**
Drawing(s)	**11** +11

10. If you are also filing any of the following, state how many against each item

Priority documents	-
Translation of priority documents	-
Statement of inventorship and right to grant a patent (PF 7/77)	-
Request for a preliminary examination and search (PF 9/77)	-
Request for substantive examination (PF 10/77)	-
Any other documents (please specify)	-

11. We request the grant of a patent on the basis of this application.

Signature: **APPLEYARD LEES** Date: **14 July 2003**

12. Contact **Ian Robinson- 01422 330110**

METHOD AND APPARATUS FOR PARTITIONING CODE IN PROGRAM CODE CONVERSION

The subject invention relates generally to the field of computers and computer software and, more particularly, to program code conversion methods and apparatus useful, for example, in code translators, emulators and accelerators.

In both embedded and non-embedded CPU's, one finds predominant Instruction Set Architectures (ISAs) for which large bodies of software exist that could be "accelerated" for performance, or "translated" to a myriad of capable processors that could present better cost/performance benefits, provided that they could transparently access the relevant software. One also finds dominant CPU architectures that are locked in time to their ISA, and cannot evolve in performance or market reach. Such architectures would benefit from "Synthetic CPU" co-architecture.

Program code conversion methods and apparatus facilitate such acceleration, translation and co-architecture capabilities and are addressed, for example, in WO 99/03168 entitled Program Code Conversion.

During program code conversion of a subject program designed for a subject architecture to a target program executable by a target architecture, a problem arises with respect to code that is self-modifying. "Self-modifying code" refers to a subject program that intentionally modifies its own subject code. There are several reasons

why a program might modify its own code, where some examples of self-modifying code are listed in Table 1.

Code	Code Function
Overlays	Overlays are a mechanism used by systems that do not support virtual memory. To save address space, a single process can re-use a subject address range to hold different libraries at different times. Such uses may or may not be associated with system calls to mmap() and munmap().
Trampolines	A trampoline is a short section of code constructed in the data area (i.e., on the stack or in the heap) that contains a call to code elsewhere in the system.
Code Patching	Linkers and debuggers may modify (patch) existing code to implement linking or breakpoint operations.
Run-Time Compilers	This category includes dynamic binary translators and just-in-time (JIT) compilers. Such programs potentially write many fragments of subject code all over the data area.
Signal Handler	A signal handler for SIGILL (illegal instruction) might modify the code that caused the exception and continue.

Table 1: Examples of Self-Modifying Code

One of the main problems presented by self-modifying code to dynamic translators is that is that the subject code that was modified may correspond to target code which has already been translated. When such a modification of the subject code occurs, all translations of the modified subject code must be identified and discarded as stale. Thus, the translator must be able to identify all target code sequences (i.e., translations) that correspond to particular subject code addresses being modified. In dynamic translators, finding and deleting the target code which corresponds to a given subject address is difficult and sometimes not even possible. In some situations, optimizations are applied during translation which yield translations that can no longer be exactly correlated to the range of subject addresses that the translations

represent. In these situations, if the subject program modifies its own code at certain subject address, the translator has no way to identify which respective translated target code to invalidate.

According to the present invention there is provided an apparatus and method as set forth in the appended claims. Preferred features of the invention will be apparent from the dependent claims, and the description which follows.

The following is a summary of various aspects and advantages realizable according to various embodiments according to the invention. It is provided as an introduction to assist those skilled in the art to more rapidly assimilate the detailed design discussion that ensues and does not and is not intended in any way to limit the scope of the claims that are appended hereto.

In particular, the inventors have developed a number of optimization techniques directed at expediting program code conversion, particularly useful in connection with a run-time translator which employs translation of successive basic blocks of subject program code into target code wherein the target code corresponding to a first basic block is executed prior to generation of target code for the next basic block.

In a preferred process, the translator employs a partitioning technique to divide the subject code space into regions, referred to hereafter as partitions, where each partition contains a distinct set of basic blocks of subject code and corresponding target code. Partitioning

divides the translator's representation of subject code and subject code translations into non-overlapping regions of subject memory. In this manner, when the subject program modifies subject code, only those partitions actually affected by the self-modifying code need be discarded and all translations in unaffected partitions can be kept. This partitioning technique is advantageous in limiting the amount of target code that must be retranslated in response to self-modifying code operation. In another process, the partitioning technique allows multithreaded subject programs that also involve self-modifying code to perform code modification in a thread-safe manner.

The accompanying drawings, which are incorporated in and constitute a part of the specification, illustrate presently preferred implementations and are described as follows:

Figure 1 is a block diagram of apparatus wherein embodiments of the invention find application;

Figure 2 is a schematic diagram illustrating a run-time translation process and corresponding IR (intermediate representation) generated during the process;

Figure 3 is a schematic illustrating a basic block data structure and cache according to an illustrative embodiment of the invention;

Figure 4 is a flow diagram illustrating an extended basic block process;

Figure 5 is a flow diagram illustrating isoblocking;

Figure 6 is a flow diagram illustrating group blocking and attendant optimizations;

Figure 7 is a schematic diagram of an example illustrating group block optimization;

Figure 8 is a flow diagram illustrating run-time translation, including extended basic blocking, isoblocking, and group blocking;

Figure 9 is a schematic diagram of an example illustrating partitioning employed by the translator of the present invention;

Figure 10 is a schematic diagram of an example illustrating reducible partitions employed by the translator of the present invention; and

Figure 11 is a schematic diagram of an example illustrating control flow between partitions by the translator of the present invention.

Illustrative apparatus for implementing various novel features discussed below is shown in Figure 1. Figure 1 illustrates a target processor 13 including target registers 15 together with memory 18 storing a number of software components 19, 20, 21, and providing working storage 16 including a basic block cache 23, a global register store 27, and the subject code 17 to be translated. The software components include an operating

system 20, the translator code 19, and translated code 21. The translator code 19 may function, for example, as an emulator translating subject code of one ISA into translated code of another ISA or as an accelerator for translating subject code into translated code, each of the same ISA.

The translator 19, i.e., the compiled version of the source code implementing the translator, and the translated code 21, i.e., the translation of the subject code 17 produced by the translator 19, run in conjunction with the operating system 20 such as, for example, UNIX running on the target processor 13, typically a microprocessor or other suitable computer. It will be appreciated that the structure illustrated in Figure 1 is exemplary only and that, for example, software, methods and processes according to the invention may be implemented in code residing within or beneath an operating system. The subject code, translator code, operating system, and storage mechanisms may be any of a wide variety of types, as known to those skilled in the art.

In apparatus according to Figure 1, program code conversion is preferably performed dynamically, at run-time, while the translated code 21 is running. The translator 19 runs inline with the translated program 21. The execution path of the translation process is a control loop comprising the steps of: executing translator code 19, which translates a block of the subject code 17 into translated code 21, and then executing that block of translated code; the end of each block of translated code contains instructions to return control back to the

translator code 19. In other words, the steps of translating and then executing the subject code are interlaced, such that only portions of the subject program 17 are translated at a time and the translated code of a first basic block is executed prior to the translation of subsequent basic blocks. The translator's fundamental unit of translation is the basic block, meaning that the translator 19 translates the subject code 17 one basic block at a time. A basic block is formally defined as a section of code with exactly one entry point and exactly one exit point, which limits the block code to a single control path. For this reason, basic blocks are the fundamental unit of control flow.

In the process of generating the translated code 21, intermediate representation ("IR") trees are generated based on the subject instruction sequence. IR trees are abstract representations of the expressions calculated and operations performed by the subject program. Later, translated code 21 is generated based on the IR trees.

The collections of IR nodes described herein are colloquially referred to as "trees". We note that, formally, such structures are in fact directed acyclic graphs (DAGs), not trees. The formal definition of a tree requires that each node have at most one parent. Because the embodiments described use common subexpression elimination during IR generation, nodes will often have multiple parents. For example, the IR of a flag-affecting instruction result may be referred to by two abstract registers, those corresponding to the destination subject register and the flag result parameter.

For example, the subject instruction "*add %r1, %r2, %r3*" performs the addition of the contents of subject registers %r2 and %r3 and stores the result in subject register %r1. Thus, this instruction corresponds to the abstract expression "%r1 = %r2 + %r3". This example contains a definition of the abstract register %r1 with an add expression containing two subexpressions representing the instruction operands %r2 and %r3. In the context of a subject program 17, these subexpressions may correspond to other, prior subject instructions, or they may represent details of the current instruction such as immediate constant values.

When the "add" instruction is parsed, a new "+" IR node is generated, corresponding to the abstract mathematical operator for addition. The "+" IR node stores references to other IR nodes that represent the operands (represented in the IR as subexpression trees, often held in subject registers). The "+" node is itself referenced by the subject register whose value it defines (the abstract register for %r1, the instruction's destination register). For example, the center-right portion of Figure 20 shows the IR tree corresponding to the X86 instruction "add %ecx, %edx".

As those skilled in the art may appreciate, in one embodiment the translator 19 is implemented using an object-oriented programming language such as C++. For example, an IR node is implemented as a C++ object, and references to other nodes are implemented as C++ references to the C++ objects corresponding to those other nodes. An IR tree is therefore implemented as a

collection of IR node objects, containing various references to each other.

Further, in the embodiment under discussion, IR generation uses a set of abstract registers. These abstract registers correspond to specific features of the subject architecture. For example, there is a unique abstract register for each physical register on the subject architecture ("subject register"). Similarly, there is a unique abstract register for each condition code flag present on the subject architecture. Abstract registers serve as placeholders for IR trees during IR generation. For example, the value of subject register %r2 at a given point in the subject instruction sequence is represented by a particular IR expression tree, which is associated with the abstract register for subject register %r2. In one embodiment, an abstract register is implemented as a C++ object, which is associated with a particular IR tree via a C++ reference to the root node object of that tree.

In the example instruction sequence described above, the translator has already generated IR trees corresponding to the values of %r2 and %r3 while parsing the subject instructions that precede the "add" instruction. In other words, the subexpressions that calculate the values of %r2 and %r3 are already represented as IR trees. When generating the IR tree for the "add %r1, %r2, %r3" instruction, the new "+" node contains references to the IR subtrees for %r2 and %r3.

The implementation of the abstract registers is divided between components in both the translator code 19

and the translated code 21. Within the translator 19, an "abstract register" is a placeholder used in the course of IR generation, such that the abstract register is associated with the IR tree that calculates the value of the subject register to which the particular abstract register corresponds. As such, abstract registers in the translator may be implemented as a C++ object which contains a reference to an IR node object (i.e., an IR tree). The aggregate of all IR trees referred to by the abstract register set is referred to as the working IR forest ("forest" because it contains multiple abstract register roots, each of which refers to an IR tree). The working IR forest represents a snapshot of the abstract operations of the subject program at a particular point in the subject code.

Within the translated code 21, an "abstract register" is a specific location within the global register store, to and from which subject register values are synchronized with the actual target registers. Alternatively, when a value has been loaded from the global register store, an abstract register in the translated code 21 could be understood to be a target register 15, which temporarily holds a subject register value during the execution of the translated code 21, prior to being saved back to the register store.

An example of program translation as described above is illustrated in Figure 2. Figure 2 shows the translation of two basic blocks of x86 instructions, and the corresponding IR trees that are generated in the process of translation. The left side of Figure 2 shows the execution path of the translator 19 during

translation. In step 151, the translator 19 translates a first basic block 153 of subject code into target code 21 and then, in step 155, executes that target code 21. When the target code 21 finishes execution, control is returned to the translator 19, step 157, wherein the translator translates the next basic block 159 of subject code 17 into target code 21 and then executes that target code 21, step 161, and so on.

In the course of translating the first basic block 153 of subject code into target code, the translator 19 generates an IR tree 163 based on that basic block 153. In this case, the IR tree 163 is generated from the source instruction "*add %ecx, %edx,*" which is a flag-affecting instruction. In the course of generating the IR tree 163, four abstract registers are defined by this instruction: the destination abstract register %ecx 167, the first flag-affecting instruction parameter 169, the second flag-affecting instruction parameter 171, and the flag-affecting instruction result 173. The IR tree corresponding to the "add" instruction is a "+" operator 175 (i.e., arithmetic addition), whose operands are the subject registers %ecx 177 and %edx 179.

Thus, emulation of the first basic block 153 puts the flags in a pending state by storing the parameters and result of the flag-affecting instruction. The flag-affecting instruction is "add %ecx, %edx." The parameters of the instruction are the current values of emulated subject registers %ecx 177 and %edx 179. The "@" symbol preceding the subject register uses 177, 179 indicate that the values of the subject registers are retrieved from the global register store, from the locations corresponding to

%ecx and %edx, respectively, as these particular subject registers were not previously loaded by the current basic block. These parameter values are then stored in the first and second flag parameter abstract registers 169, 171. The result of the addition operation 175 is stored in the flag result abstract register 173.

After the IR tree is generated, the corresponding target code 21 is generated based on the IR. The process of generating target code 21 from a generic IR is well understood in the art. Target code is inserted at the end of the translated block to save the abstract registers, including those for the flag result 173 and the flag parameters 169, 171, to the global register store 27. After the target code is generated, it is then executed, step 155.

Figure 2 shows an example of translation and execution interlaced. The translator 19 first generates translated code 21 based on the subject instructions 17 of a first basic block 153, then the translated code for basic block 153 is executed. At the end of the first basic block 153, the translated code 21 returns control to the translator 19, which then translates a second basic block 159. The translated code 21 for the second basic block 161 is then executed. At the end of the execution of the second basic block 159, the translated code returns control to the translator 19, which then translates the next basic block, and so forth.

Thus, a subject program running under the translator 19 has two different types of code that execute in an interleaved manner: the translator code 19 and the

translated code 21. The translator code 19 is generated by a compiler, prior to run-time, based on the high-level source code implementation of the translator 19. The translated code 21 is generated by the translator code 19, throughout run-time, based on the subject code 17 of the program being translated.

The representation of the subject processor state is likewise divided between the translator 19 and translated code 21 components. The translator 19 stores subject processor state in a variety of explicit programming language devices such as variables and/or objects; the compiler used to compile the translator determines how the state and operations are implemented in target code. The translated code 21, by comparison, stores subject processor state implicitly in target registers and memory locations, which are manipulated directly by the target instructions of the translated code 21.

For example, the low-level representation of the global register store 27 is simply a region of allocated memory. This is how the translated code 21 sees and interacts with the abstract registers, by saving and restoring between the defined memory region and various target registers. In the source code of the translator 19, however, the global register store 27 is a data array or an object which can be accessed and manipulated at a higher level. With respect to the translated code 21, there simply is no high-level representation.

In some cases, subject processor state which is static or statically determinable in the translator 19 is encoded directly into the translated code 21 rather than being

calculated dynamically. For example, the translator 19 may generate translated code 21 that is specialized on the instruction type of the last flag-affecting instruction, meaning that the translator would generate different target code for the same basic block if the instruction type of the last flag-affecting instruction changed.

The translator 19 contains data structures corresponding to each basic block translation, which particularly facilitates extended basic block, isoblock, group block, and cached translation state optimizations as hereafter described. Figure 3 illustrates such a basic block data structure 30, which includes a subject address 31, a target code pointer 33 (i.e., the target address of the translated code), translation hints 34, entry and exit conditions 35, a profiling metric 37, references to the data structures of the predecessor and successor basic blocks 38, 39, and an entry register map 40. Figure 3 further illustrates the basic block cache 23, which is a collection of basic block data structures, e.g., 30, 41, 42, 43, 44 . . . indexed by subject address. In one embodiment, the data corresponding to a particular translated basic block may be stored in a C++ object. The translator creates a new basic block object as the basic block is translated.

The subject address 31 of the basic block is the starting address of that basic block in the memory space of the subject program 17, meaning the memory location where the basic block would be located if the subject program 17 were running on the subject architecture. This is also referred to as the subject starting address. While each basic block corresponds to a range of subject

addresses (one for each subject instruction), the subject starting address is the subject address of the first instruction in the basic block.

The target address 33 of the basic block is the memory location (starting address) of the translated code 21 in the target program. The target address 33 is also referred to as the target code pointer, or the target starting address. To execute a translated block, the translator 19 treats the target address as a function pointer which is dereferenced to invoke (transfer control to) the translated code.

The basic block data structures 30, 41, 42, 43, ... are stored in the basic block cache 23, which is a repository of basic block objects organized by subject address. When the translated code of a basic block finishes executing, it returns control to the translator 19 and also returns the value of the basic block's destination (successor) subject address 31 to the translator. To determine if the successor basic block has already been translated, the translator 19 compares the destination subject address 31 against the subject addresses 31 of basic blocks in the basic block cache 23 (i.e., those that have already been translated). Basic blocks which have not been yet translated are translated and then executed. Basic blocks which have already been translated (and which have compatible entry conditions, as discussed below) are simply executed. Over time, many of the basic blocks encountered will already have been translated, which causes the incremental translation cost to decrease. As such, the translator 19 gets faster over time, as fewer and fewer blocks require translation.

Extended Basic Blocks

One optimization applied according to the illustrative embodiment is to increase the scope of code generation by a technique referred to as "extended basic blocks." In cases where a basic block A has only one successor block (e.g., basic block B), the translator may be able to statically determine (when A is decoded) the subject address of B. In such cases, basic blocks A and B are combined into a single block (A') which is referred to as an extended basic block. Put differently, the extended basic block mechanism can be applied to unconditional jumps whose destination is statically determinable; if a jump is conditional or if the destination cannot be statically determined, then a separate basic block must be formed. An extended basic block may still formally be a basic block, because after the intervening jump from A to B is removed, the code of block A' has only a single flow of control, and therefore no synchronization is necessary at the AB boundary.

Even if A has multiple possible successors including B, extended basic blocks may be used to extend A into B for a particular execution in which B is the actual successor and B's address is statically determinable.

Statically determinable addresses are those the translator can determine at decode-time. During construction of a block's IR forest, an IR tree is constructed for the destination subject address, which is associated with the destination address abstract register. If the value of destination address IR tree is statically

determinable (i.e., does not depend on dynamic or run-time subject register values), then the successor block is statically determinable. For example, in the case of an unconditional jump instruction, the destination address (i.e., the subject starting address of the successor block) is implicit in the jump instruction itself; the subject address of the jump instruction plus the offset encoded in the jump instruction equals the destination address. Likewise, the optimizations of constant folding (e.g., X + (2 + 3) => X + 5) and expression folding (e.g., (X * 5) * 10 => X * 50) may cause an otherwise "dynamic" destination address to become statically determinable. The calculation of the destination address thus consists of extracting the constant value from the destination address IR.

When extended basic block A' is created, the translator subsequently treats it the same as any other basic block when performing IR generation, optimizations, and code generation. Because the code generation algorithms are operating on a larger scope (i.e., the code of basic blocks A and B combined), the translator 19 generates more optimal code.

As one of ordinary skill in the art will appreciate, decoding is the process of extracting individual subject instructions from the subject code. The subject code is stored as an unformatted byte stream (i.e., a collection of bytes in memory). In the case of subject architectures with variable-length instructions (e.g., X86), decoding first requires the identification of instruction boundaries; in the case of fixed-length instruction architectures, identifying instruction boundaries is

trivial (e.g., on the MIPS, every four bytes is an instruction). The subject instruction format is then applied to the bytes that constitute a given instruction to extract the instruction data (i.e., the instruction type, operand register numbers, immediate field values, and any other information encoded in the instruction). The process of decoding machine instructions of a known architecture from an unformatted byte stream using that architecture's instruction format is well understood in the art.

Figure 4 illustrates the creation of an extended basic block. A set of constituent basic blocks which is eligible to become an extended basic block is detected when the earliest eligible basic block (A) is decoded. If the translator 19 detects that A's successor (B) is statically determinable 51, it calculates B's starting address 53 and then resumes the decoding process at the starting address of B. If B's successor (C) is determined to be statically determinable 55, the decoding process proceeds to the starting address of C, and so forth. Of course, if a successor block is not statically determinable then normal translation and execution resume 61, 63, 65.

During all basic block decoding, the working IR forest includes an IR tree to calculate the subject address 31 of the current block's successor (i.e., the destination subject address; the translator has a dedicated abstract register for the destination address). In the case of an extended basic block, to compensate for the fact that intervening jumps are being eliminated, as each new constituent basic block is assimilated by the decoding

process, the IR tree for the calculation of that block's subject address is pruned 54 (Figure 4). In other words, when the translator 19 statically calculates B's address and decoding resumes at B's starting address, the IR tree corresponding to the dynamic calculation of B's subject address 31 (which was constructed in the course of decoding A) is pruned; when decoding proceeds to the starting address of C, the IR tree corresponding to C's subject address is pruned 59; and so forth. "Pruning" an IR tree means to remove any IR nodes which are depended on by the destination address abstract register and by no other abstract registers. Put differently, pruning breaks the link between the IR tree and the destination abstract register; any other links to the same IR tree remain unaffected. In some cases, a pruned IR tree may also be depended on by another abstract register, in which case the IR tree remains to preserve the subject program's execution semantics.

To prevent code explosion (traditionally, the mitigating factor against such code specialization techniques), the translator limits extended basic blocks to some maximum number of subject instructions. In one embodiment, extended basic blocks are limited to a maximum of 200 subject instructions.

Isoblocks

Another optimization implemented in the illustrated embodiment is so-called "isoblocking." According to this technique, translations of basic blocks are parameterized, or specialized, on a compatibility list, which is a set of variable conditions that describe the subject processor

state and the translator state. The compatibility list is different for each subject architecture, to take into account different architectural features. The actual values of the compatibility conditions at the entry and exit of a particular basic block translation are referred to as entry conditions and exit conditions, respectively.

If execution reaches a basic block which has already been translated but the previous translation's entry conditions differ from the current working conditions (i.e., the exit conditions of the previous block), then the basic block must be translated again, this time based on the current working conditions. The result is that the same subject code basic block is now represented by multiple target code translations. These different translations of the same basic block are referred to as isoblocks.

To support isoblocks, the data associated with each basic block translation includes one set of entry conditions 35 and one set of exit conditions 36 (Figure 3). In one embodiment, the basic block cache 23 is organized first by subject address 31 and then by entry conditions 35, 36 (Figure 3). In another embodiment, when the translator queries the basic block cache 23 for a subject address 31, the query may return multiple translated basic blocks (isoblocks).

Figure 5 illustrates the use of isoblocks. At the end of a first translated block's execution, the translated code 21 calculates and returns the subject address of the next block (i.e., the successor) 71. Control is then returned to the translator 19, as demarcated by dashed

line 73. In the translator 19, the basic block cache 23 is queried using the returned subject address 31, step 75. The basic block cache may return zero, one, or more than one basic block data structures with the same subject address 31. If the basic block cache 23 returns zero data structures (meaning that this basic block has not yet been translated), then the basic block must be translated, step 77, by the translator 19. Each data structure returned by the basic block cache 23 corresponds to a different translation (isoblock) of the same basic block of subject code. As illustrated at decision diamond 79, if the current exit conditions (of the first translated block) do not match the entry conditions of any of the data structures returned by the basic block cache 23, then the basic block must be translated again, step 81, this time parameterized on those exit conditions. If the current exit conditions match the entry conditions of one of the data structures returned by the basic block cache 23, then that translation is compatible and can be executed without re-translation, step 83. In the illustrative embodiment, the translator 19 executes the compatible translated block by dereferencing the target address as a function pointer.

As noted above, basic block translations are preferably parameterized on a compatibility list. Exemplary compatibility lists will now be described for both the X86 and PowerPC architectures.

An illustrative compatibility list for the X86 architecture includes representations of: (1) lazy propagation of subject registers; (2) overlapping abstract registers; (3) type of pending condition code flag-affecting instruction; (4) lazy propagation of condition

code flag-affecting instruction parameters; (5) direction of string copy operations; (6) floating point unit (FPU) mode of the subject processor; and (7) modifications of the segment registers.

The compatibility list for the X86 architecture includes representations of any lazy propagation of subject registers by the translator, also referred to as register aliasing. Register aliasing occurs when the translator knows that two subject registers contain the same value at a basic block boundary. As long as the subject register values remain the same, only one of the corresponding abstract registers is synchronized, by saving it to the global register store. Until the saved subject register is overwritten, references to the non-saved register simply use or copy (via a move instruction) the saved register. This avoids two memory accesses (save + restore) in the translated code.

The compatibility list for the X86 architecture includes representations of which of the overlapping abstract registers are currently defined. In some cases, the subject architecture contains multiple overlapping subject registers which the translator represents using multiple overlapping abstract registers. For example, variable-width subject registers are represented using multiple overlapping abstract registers, one for each access size. For example, the X86 "EAX" register can be accessed using any of the following subject registers, each of which has a corresponding abstract register: EAX (bits 31…0), AX (bits 15…0), AH (bits 15…8), and AL (bits 7…0).

The compatibility list for the X86 architecture includes representations of, for each integer and floating point condition code flag, whether the flag value is normalized or pending, and if pending the type of the pending flag-affecting instruction.

The compatibility list for the X86 architecture includes representations of register aliasing for condition code flag-affecting instruction parameters (if some subject register still holds the value of a flag-affecting instruction parameter, or if the value of the second parameter is the same as the first). The compatibility list also includes representations of whether the second parameter is a small constant (i.e., an immediate instruction candidate), and if so its value. The compatibility list for the X86 architecture includes a representation of the current direction of string copy operations in the subject program. This condition field indicates whether string copy operations move upward or downward in memory. This supports code specialization of "strcpy()" function calls, by parameterizing translations on the function's direction argument.

The compatibility list for the X86 architecture includes a representation of the FPU mode of the subject processor. The FPU mode indicates whether subject floating-point instructions are operating in 32- or 64-bit mode.

The compatibility list for the X86 architecture includes a representation of modifications of the segment registers. All X86 instruction memory references are based on one of six memory segment registers: CS (code

segment), DS (data segment), SS (stack segment), ES (extra data segment), FS (general purpose segment), and GS (general purpose segment). Under normal circumstances an application will not modify the segment registers. As such, code generation is by default specialized on the assumption that the segment register values remain constant. It is possible, however, for a program to modify its segment registers, in which case the corresponding segment register compatibility bit will be set, causing the translator to generate code for generalized memory accesses using the appropriate segment register's dynamic value.

An illustrative embodiment of a compatibility list for the PowerPC architecture includes representations of: (1) mangled registers; (2) link value propagation; (3) type of pending condition code flag-affecting instruction; (4) lazy propagation of condition code flag-affecting instruction parameters; (5) condition code flag value aliasing; and (6) summary overflow flag synchronization state.

The compatibility list for the PowerPC architecture includes a representation of mangled registers. In cases where the subject code contains multiple consecutive memory accesses using a subject register for the base address, the translator may translate those memory accesses using a mangled target register. In cases where subject program data is not located at the same address in target memory as it would have been in subject memory, the translator must include a target offset in every memory address calculated by the subject code. While the subject register contains the subject base address, a mangled

target register contains the target address corresponding to that subject base address (i.e., subject base address + target offset). With register mangling, memory accesses can be translated more efficiently by applying the subject code offsets directly to the target base address, stored in the mangled register. By comparison, without the mangled register mechanism this scenario would require additional manipulation of the target code for each memory access, at the cost of both space and execution time. The compatibility list indicates which abstract registers if any are mangled.

The compatibility list for the PowerPC architecture includes a representation of link value propagation. For leaf functions (i.e., functions that call no other functions), the function body may be extended (as with the extended basic block mechanism discussed above) into the call/return site. Hence, the function body and the code that follows the function's return are translated together. This is also referred to as function return specialization, because such a translation includes code from, and is therefore specialized on, the function's return site. Whether a particular block translation used link value propagation is reflected in the exit conditions. As such, when the translator encounters a block whose translation used link value propagation, it must evaluate whether the current return site will be the same as the previous return site. Functions return to the same location from which they are called, so the call site and return site are effectively the same (offset by one or two instructions). The translator can therefore determine whether the return sites are the same by comparing the respective call sites; this is equivalent to comparing the

subject addresses of the respective predecessor blocks (of the function block's prior and current executions). As such, in embodiments that support link value propagation, the data associated with each basic block translation includes a reference to the predecessor block translation (or some other representation of the predecessor block's subject address).

The compatibility list for the PowerPC architecture includes representations of, for each integer and floating point condition code flag, whether the flag value is normalized or pending, and if pending the type of the pending flag-affecting instruction.

The compatibility list for the PowerPC architecture includes representations of register aliasing for flag-affecting instruction parameters (if flag-affecting instruction parameter values happen to be live in a subject register, or if the value of the second parameter is the same as the first). The compatibility list also includes representations of whether the second parameter is a small constant (i.e., an immediate instruction candidate), and if so its value.

The compatibility list for the PowerPC architecture includes representations of register aliasing for the PowerPC condition code flag values. The PowerPC architecture includes instructions for explicitly loading the entire set of PowerPC flags into a general purpose (subject) register. This explicit representation of the subject flag values in subject registers interferes with the translator's condition code flag emulation optimizations. The compatibility list contains a

representation of whether the flag values are live in a subject register, and if so which register. During IR generation, references to such a subject register while it holds the flag values are translated into references to the corresponding abstract registers. This mechanism eliminates the need to explicitly calculate and store the subject flag values in a target register, which in turn allows the translator to apply the standard condition code flag optimizations.

The compatibility list for the PowerPC architecture includes a representation of summary overflow synchronization. This field indicates which of the eight summary overflow condition bits are current with the global summary overflow bit. When one of the PowerPC's eight condition fields is updated, if the global summary overflow is set, it is copied to the corresponding summary overflow bit in the particular condition code field.

Translation Hints

Another optimization implemented in the illustrative embodiment employs the translation hints 34 of the basic block data structure of Figure 3. This optimization proceeds from a recognition that there is static basic block data which is specific to a particular basic block, but which is the same for every translation of that block. For some types of static data which are expensive to calculate, it is more efficient for the translator to calculate the data once, during the first translation of the corresponding block, and then store the result for future translations of the same block. Because this data is the same for every translation of the same block, it

does not parameterize translation and therefore it is not formally part of the block's compatibility list (discussed above). Expensive static data is still stored in the data associated with each basic block translation, however, as it is cheaper to save the data than it is to recalculate. In later translations of the same block, even if the translator 19 cannot reuse a prior translation, the translator 19 can take advantage of these "translation hints" (i.e., the cached static data) to reduce the translation cost of the second and later translations.

In one embodiment, the data associated with each basic block translation includes translation hints, which are calculated once during the first translation of that block and then copied (or referred to) on each subsequent translation.

For example, in a translator 19 implemented in C++, translation hints may be implemented as a C++ object, in which case the basic block objects which correspond to different translations of the same block would each store a reference to the same translation hints object. Alternatively, in a translator implemented in C++, the basic block cache 23 may contain one basic block object per subject basic block (rather than per translation), with each such object containing or holding a reference to the corresponding translation hints; such basic block objects also contain multiple references to translation objects that correspond to different translations of that block, organized by entry conditions.

Exemplary translation hints for the X86 architecture include representations of: (1) initial instruction

prefixes; and (2) initial repeat prefixes. Such translation hints for the X86 architecture particularly include a representation of how many prefixes the first instruction in the block has. Some X86 instructions have prefixes which modify the operation of the instruction. This architectural feature makes it difficult (i.e., expensive) to decode an X86 instruction stream. Once the number of initial prefixes is determined during the first decoding of the block, that value is then stored by the translator 19 as a translation hint, so that subsequent translations of the same bock do not need to determine it anew.

The translation hints for the X86 architecture further include a representation of whether the first instruction in the block has a repeat prefix. Some X86 instructions such as string operations have a repeat prefix which tells the processor to execute that instruction multiple times. The translation hints indicate whether such a prefix is present, and if so its value.

In one embodiment, the translation hints associated with each basic block additionally include the entire IR forest corresponding to that basic block. This effectively caches all of the decoding and IR generation performed by the frontend. In another embodiment, the translation hints include the IR forest as it exists prior to being optimized. In another embodiment, the IR forest is not cached as a translation hint, in order to conserve the memory resources of the translated program.

Group Blocks

Another optimization implemented in the illustrative translator embodiment is directed to eliminating program overhead resulting from the necessity to synchronize all abstract registers at the end of execution of each translated basic block. This optimization is referred to as group block optimization.

As discussed above, in basic block mode (e.g., Figure 2), state is passed from one basic block to the next using a memory region which is accessible to all translated code sequences, namely, a global register store 27. The global register store 27 is a repository for abstract registers, each of which corresponds to and emulates the value of a particular subject register or other subject architectural feature. During the execution of translated code 21, abstract registers are held in target registers so that they may participate in instructions. During the execution of translator code 21, abstract register values are stored in the global register store 27 or target registers 15.

Thus, in basic block mode such as illustrated in Figure 2, all abstract registers must be synchronized at the end of each basic block for two reasons: (1) control returns to the translator code 19, which potentially overwrites all target registers; and (2) because code generation only sees one basic block at a time, the translator 19 must assume that all abstract registers values are live (i.e., will be used in subsequent basic blocks) and therefore must be saved. The goal of the group block optimization mechanism is to reduce

synchronization across basic block boundaries that are crossed frequently, by translating multiple basic blocks as a contiguous whole. By translating multiple basic blocks together, the synchronization at block boundaries can be minimized if not eliminated.

Group block construction is triggered when the current block's profiling metric reaches a trigger threshold. This block is referred to as the trigger block. Construction can be separated into the following steps (Figure 6): (1) selecting member blocks 71; (2) ordering member blocks 73; (3) global dead code elimination 75; (4) global register allocation 77; and (5) code generation 79. The first step 71 identifies the set of blocks that are to be included in the group block by performing a depth-first search (DFS) traversal of the program's control flow graph, beginning with the trigger block and tempered by an inclusion threshold and a maximum member limit. The second step 73 orders the set of blocks and identifies the critical path through the group block, to enable efficient code layout that minimizes synchronization code and reduces branches. The third and fourth steps 75, 77 perform optimizations. The final step 79 generates target code for all member blocks in turn, producing efficient code layout with efficient register allocation.

In construction of a group block and generation of target code therefrom, the translator code 19 implements the steps illustrated in Figure 6. When the translator 19 encounters a basic block that was previously translated, prior to executing that block, the translator 19 checks the block's profiling metric 37 (Figure 3) against the trigger threshold. The translator 19 begins group block

creation when a basic block's profiling metric 37 exceeds the trigger threshold. The translator 19 identifies the members of the group block by a traversal of the control flow graph, starting with the trigger block and tempered by the inclusion threshold and maximum member limit. Next, the translator 19 creates an ordering of the member blocks, which identifies the critical path through the group block. The translator 19 then performs global dead code elimination; the translator 19 gathers register liveness information for each member block, using the IR corresponding to each block. Next, the translator 19 performs global register allocation according to an architecture-specific policy, which defines a partial set of uniform register mappings for all member blocks. Finally, the translator 19 generates target code for each member block in order, consistent with the global register allocation constraints and using the register liveness analyses.

As noted above, the data associated with each basic block includes a profiling metric 37. In one embodiment, the profiling metric 37 is execution count, meaning that the translator 19 counts the number of times a particular basic block has been executed; in this embodiment, the profiling metric 37 is represented as an integer count field (counter). In another embodiment, the profiling metric 37 is execution time, meaning that the translator 19 keeps a running aggregate of the execution time for all executions of a particular basic block, such as by planting code in the beginning and end of a basic block to start and stop, respectively, a hardware or software timer; in this embodiment, the profiling metric 37 uses some representation of the aggregate execution time

(timer). In another embodiment, the translator 19 stores multiple types of profiling metrics 37 for each basic block. In another embodiment, the translator 19 stores multiple sets of profiling metrics 37 for each basic block, corresponding to each predecessor basic block and/or each successor basic block, such that distinct profiling data is maintained for different control paths. In each translator cycle (i.e., the execution of translator code 19 between executions of translated code 21), the profiling metric 37 for the appropriate basic block is updated.

In embodiments that support group blocks, the data associated with each basic block additionally includes references 38, 39 to the basic block objects of known predecessors and successors. These references in aggregate constitute a control-flow graph of all previously executed basic blocks. During group block formation, the translator 19 traverses this control-flow graph to determine which basic blocks to include in the group block under formation.

Group block formation in the illustrative embodiment is based on three thresholds: a trigger threshold, an inclusion threshold, and a maximum member limit. The trigger threshold and the inclusion threshold refer to the profiling metric 37 for each basic block. In each translator cycle, the profiling metric 37 of the next basic block is compared to the trigger threshold. If the metric 37 meets the trigger threshold then group block formation begins. The inclusion threshold is then used to determine the scope of the group block, by identifying which successor basic blocks to include in the group

block. The maximum member limit defines the upper limit on the number of basic blocks to be included in any one group block.

When the trigger threshold is reached for basic block A, a new group block is formed with A as the trigger block. The translator 19 then begins the definition traversal, a traversal of A's successors in the control-flow graph to identify other member blocks to include. When traversal reaches a given basic block, its profiling metric 37 is compared to the inclusion threshold. If the metric 37 meets the inclusion threshold, that basic block is marked for inclusion and the traversal continues to the block's successors. If the block's metric 37 is below the inclusion threshold, that block is excluded and its successors are not traversed. When traversal ends (i.e., all paths either reach an excluded block or cycle back to an included block, or the maximum member limit is reached), the translator 19 constructs a new group block based on all of the included basic blocks.

In embodiments that use isoblocks and group blocks, the control flow graph is a graph of isoblocks, meaning that different isoblocks of the same subject block are treated as different blocks for the purposes of group block creation. Thus, the profiling metrics for different isoblocks of the same subject block are not aggregated.

In another embodiment, isoblocks are not used in basic block translation but are used in group block translation, meaning that non-group basic block translations are generalized (not specialized on entry conditions). In this embodiment, a basic block's profiling metric is

disaggregated by the entry conditions of each execution, such that distinct profiling information is maintained for each theoretical isoblock (i.e., for each distinct set of entry conditions). In this embodiment, the data associated with each basic block includes a profiling list, each member of which is a three-item set containing: (1) a set of entry conditions, (2) a corresponding profiling metric, and (3) a list of corresponding successor blocks. This data maintains profiling and control path information for each set of entry conditions to the basic block, even though the actual basic block translation is not specialized on those entry condition. In this embodiment, the trigger threshold is compared to each profiling metric within a basic block's profiling metric list. When the control flow graph is traversed, each element in a given basic block's profiling list is treated as a separate node in the control flow graph. The inclusion threshold is therefore compared against each profiling metric in the block's profiling list. In this embodiment, group blocks are created for particular hot isoblocks (specialized to particular entry conditions) of hot subject blocks, but other isoblocks of those same subject blocks are executed using the general (non-isoblock) translations of those blocks.

After the definition traversal, the translator 19 performs an ordering traversal, step 73; Figure 6, to determine the order in which member blocks will be translated. The order of the member blocks affects both the instruction cache behavior of the translated code 21 (hot paths should be contiguous) and the synchronization necessary on member block boundaries (synchronization should be minimized along hot paths). In one embodiment,

the translator 19 performs the ordering traversal using an ordered depth-first search (DFS) algorithm, ordered by execution count. Traversal starts at the member block having the highest execution count. If a traversed member block has multiple successors, the successor with the higher execution count is traversed first.

One of ordinary skill in the art will appreciate that group blocks are not formal basic blocks, as they may have internal control branches, multiple entry points, and/or multiple exit points.

Once a group block has been formed, a further optimization may be applied to it, referred to herein as "global dead code elimination." Such global dead code elimination employs the technique of liveness analysis. Global dead code elimination is the process of removing redundant work from the IR across a group of basic blocks.

Generally, subject processor state must be synchronized on translation scope boundaries. A value, such as a subject register, is said to be "live" for the range of code starting with its definition and ending with its last use prior to being re-defined (overwritten); hence, the analysis of values' (e.g., temporary values in the context of IR generation, target registers in the context of code generation, or subject registers in the context of translation) uses and definitions is known in the art as liveness analysis. Whatever knowledge (i.e., liveness analysis) the translator has regarding the uses (reads) and definitions (writes) of data and state is limited to its translation scope; the rest of the program is an unknown. More specifically, because the translator

does not know which subject registers will be used outside the scope of translation (e.g., in a successor basic block), it must assume that all registers will be used. As such, the values (definitions) of any subject registers which were modified within a given basic block must be saved (stored to the global register store 27) at the end of that basic block, against the possibility of their future use. Likewise, all subject registers whose values will be used in a given basic block must be restored (loaded from the global register store 27) at the beginning of that basic block; i.e., the translated code for a basic block must restore a given subject register prior to its first use within that basic block.

The general mechanism of IR generation involves an implicit form of "local" dead code elimination, whose scope is localized to only a small group of IR nodes at once. For example, a common subexpression A in the subject code would be represented by a single IR tree for A with multiple parent nodes, rather than multiple instances of the expression tree A itself. The "elimination" is implicit in the fact that one IR node can have links to multiple parent nodes. Likewise, the use of abstract registers as IR placeholders is an implicit form of dead code elimination. If the subject code for a given basic block never defines a particular subject register, then at the end of IR generation for that block, the abstract register corresponding to that subject register will refer to an empty IR tree. The code generation phase recognizes that, in this scenario, the appropriate abstract register need not be synchronized with the global register store. As such, local dead code elimination is

implicit in the IR generation phase, occurring incrementally as IR nodes are created.

In contrast to local dead code elimination, a "global" dead code elimination algorithm is applied to a basic block's entire IR expression forest. Global dead code elimination according to the illustrative embodiment requires liveness analysis, meaning analysis of subject register uses (reads) and subject register definitions (writes) within the scope of each basic block in a group block, to identify live and dead regions. The IR is transformed to remove dead regions and thereby reduce the amount of work that must be performed by the target code. For example, at a given point in the subject code, if the translator 19 recognizes or detects that a particular subject register will be defined (overwritten) before its next use, the subject register is said to be dead at all points in the code up to that preempting definition. In terms of the IR, subject registers which are defined but never used before being re-defined are dead code which can be eliminated in the IR phase without ever spawning target code. In terms of target code generation, target registers which are dead can be used for other temporary or subject register values without spilling.

In group block global dead code elimination, liveness analysis is performed on all member blocks. Liveness analysis generates the IR forest for each member block, which is then used to derive the subject register liveness information for that block. IR forests for each member block are also needed in the code generation phase of group block creation. Once the IR for each member block is generated in liveness analysis, it can either be saved

for subsequent use in code generation, or it can be deleted and re-generated during code generation.

Group block global dead code elimination can effectively "transform" the IR in two ways. First, the IR forest generated for each member block during liveness analysis can be modified, and then that entire IR forest can be propagated to (i.e., saved and reused during) the code generation phase; in this scenario, the IR transformations are propagated through the code generation phase by applying them directly to the IR forest and then saving the transformed IR forest. In this scenario, the data associated with each member block includes liveness information (to be additionally used in global register allocation), and the transformed IR forest for that block.

Alternatively and preferably, the step of global dead code elimination which transforms the IR for a member block is performed during the final code generation phase of group block creation, using liveness information created earlier. In this embodiment, the global dead code transformations can be recorded as list of "dead" subject registers, which is then encoded in the liveness information associated with each member block. The actual transformation of the IR forest is thus performed by the subsequent code generation phase, which uses the dead register list to prune the IR forest. This scenario allows the translator to generate the IR once during liveness analysis, then throw the IR away, and then re-generate the same IR during the code generation, at which point the IR is transformed using the liveness analysis (i.e., global dead code elimination is applied to the IR itself). In this scenario, the data associated with each

member block includes liveness information, which includes a list of dead subject registers. The IR forest is not saved. Specifically, after the IR forest is (re)generated in the code generation phase, the IR trees for dead subject registers (which are listed in the dead subject register list within the liveness information) are pruned.

In one embodiment, the IR created during liveness analysis is thrown away after the liveness information is extracted, to conserve memory resources. The IR forests (one per member block) are recreated during code generation, one member block at a time. In this embodiment, the IR forests for all member blocks do not coexist at any point in translation. However, the two versions of the IR forests, created during liveness analysis and code generation, respectively, are identical, as they are generated from the subject code using the same IR generation process.

In another embodiment, the translator creates an IR forest for each member block during liveness analysis, and then saves the IR forest, in the data associated with each member block, to be reused during code generation. In this embodiment, the IR forests for all member blocks coexist, from the end of liveness analysis (in the global dead code elimination step) to code generation. In one alternative of this embodiment, no transformations or optimizations are performed on the IR during the period from its initial creation (during liveness analysis) and its last use (code generation).

In another embodiment, the IR forests for all member blocks are saved between the steps of liveness analysis

and code generation, and inter-block optimizations are performed on the IR forests prior to code generation. In this embodiment, the translator takes advantage of the fact that all member block IR forests coexist at the same point in translation, and optimizations are performed across the IR forests of different member blocks which transform those IR forests. In this case, the IR forests used in code generation may not be identical to the IR forests used in liveness analysis (as in the two embodiments described above), because the IR forests have been subsequently transformed by inter-block optimizations. In other words, the IR forests used in code generation may be different than the IR forests that would result from generating them anew one member block at a time.

In group block global dead code elimination, the scope of dead code detection is increased by the fact that liveness analysis is applied to multiple blocks at the same time. Hence, if a subject register is defined in the first member block, and then redefined in the third member block (with no intervening uses or exit points), the IR tree for the first definition can be eliminated from the first member block. By comparison, under basic block code generation, the translator 19 would be unable to detect that this subject register was dead.

As noted above, one goal of group block optimization is to reduce or eliminate the need for register synchronization at basic block boundaries. Accordingly, a discussion of how register allocation and synchronization is achieved by the translator 19 during group blocking is now provided.

Register allocation is the process of associating an abstract (subject) register with a target register. Register allocation is a necessary component of code generation, as abstract register values must reside in target registers to participate in target instructions. The representation of these allocations (i.e., mappings) between target registers and abstract registers is referred to as a register map. During code generation, the translator 19 maintains a working register map, which reflects the current state of register allocation (i.e., the target-to-abstract register mappings actually in existence at a given point in the target code). Reference will be had hereafter to an exit register map which is, abstractly, a snapshot of the working register map on exit from a member block. However, since the exit register map is not needed for synchronization, it is not recorded so it is purely abstract. The entry register map 40 (Figure 3) is a snapshot of the working register map on entry to a member block, which is necessary to record for synchronization purposes.

Also, as discussed above, a group block contains multiple member blocks, and code generation is performed separately for each member block. As such, each member block has its own entry register map 40 and exit register map, which reflect the allocation of particular target registers to particular subject registers at the beginning and end, respectively, of the translated code for that block.

Code generation for a group member block is parameterized by its entry register map 40 (the working

register map on entry), but code generation also modifies the working register map. The exit register map for a member block reflects the working register map at the end of that block, as modified by the code generation process. When the first member block is translated, the working register map is empty (subject to global register allocation, discussed below). At the end of translation for the first member block, the working register map contains the register mappings created by the code generation process. The working register map is then copied into the entry register maps 40 of all successor member blocks.

At the end of code generation for a member block, some abstract registers may not require synchronization. Register maps allow the translator 19 to minimize synchronization on member block boundaries, by identifying which registers actually require synchronization. By comparison, in the (non-group) basic block scenario all abstract registers must be synchronized at the end of every basic block.

At the end of a member block, three synchronization scenarios are possible based on the successor. First, if the successor is a member block which has not yet been translated, its entry register map 40 is defined to be the same as the working register map, with the consequence that no synchronization is necessary. Second, if the successor block is external to the group, then all abstract registers must be synchronized (i.e., a full synchronization) because control will return to the translator code 19 before the successor's execution. Third, if the successor block is a member block whose

register map has already been fixed, then synchronization code must be inserted to reconcile the working map with the successor's entry map.

Some of the cost of register map synchronization is reduced by the group block ordering traversal, which minimizes register synchronization or eliminates it entirely along hot paths. Member blocks are translated in the order generated by the ordering traversal. As each member block is translated, its exit register map is propagated into the entry register map 40 of all successor member blocks whose entry register maps are not yet fixed. In effect, the hottest path in the group block is translated first, and most if not all member block boundaries along that path require no synchronization because the corresponding register maps are all consistent.

For example, the boundary between the first and second member blocks will always require no synchronization, because the second member block will always have its entry register map 40 fixed to be the same as the exit register map 41 of the first member block. Some synchronization between member blocks may be unavoidable because group blocks can contain internal control branches and multiple entry points. This means that execution may reach the same member block from different predecessors, with different working register maps at different times. These cases require that the translator 19 synchronize the working register map with the appropriate member block's entry register map.

If required, register map synchronization occurs on member block boundaries. The translator 19 inserts code at the end of a member block to synchronize the working register map with the successor's entry register map 40. In register map synchronization, each abstract register falls under one of ten synchronization conditions. Table 1 illustrates the ten register synchronization cases as a function of the translator's working register map and the successor's entry register map 40. Table 2 describes the register synchronization algorithm, by enumerating the ten formal synchronization cases with text descriptions of the cases and pseudo-code descriptions of the corresponding synchronization actions (the pseudo-code is explained below). Thus, at every member block boundary, every abstract register is synchronized using the 10-case algorithm. This detailed articulation of synchronization conditions and actions allows the translator 19 to generate efficient synchronization code, which minimizes the synchronization cost for each abstract register.

The following describes the synchronization action functions listed in Table 2. *"Spill(E(a))"* saves abstract register *a* from target register *E(a)* into the subject register bank (a component of the global register store). *"Fill(t,a)"* loads abstract register *a* from the subject register bank into target register *t*. *"Reallocate()"* moves and reallocates (i.e., changes the mapping of) an abstract register to a new target register if available, or spills the abstract register if a target register is not available. *"FreeNoSpill(t)"* marks a target register as free without spilling the associated abstract subject register. The *FreeNoSpill()* function is necessary to avoid superfluous spilling across multiple applications of

the algorithm at the same synchronization point. Note that for cases with a "Nil" synchronization action, no synchronization code is necessary for the corresponding abstract registers.

LEGEND	
a	abstract subject register
t	target register
W	working register map {W(a) => t}
E	entry register map {E(a) => t}
dom	domain
rng	range
∈	is a member of
∉	is not a member of
W(a) ∉ rng E	The working register for abstract register "a" is not in the range of the entry register map. I.e., the target register that is currently mapped to abstract register "a" ("W(a)") is not defined in the entry register map E.

	a ∈ dom W				**a ∉ dom W**
a ∈ dom E		**W(a) ∉ rng E**	**W(a) ∈ rng E**		
	E(a) ∉ rng W	6	8		4
	E(a) ∈ rng W	7	**W(a) ≠ E(a)**	9	5
			W(a) = E(a)	10	
a ∉ dom E		2	3		1

Table 1: Enumeration of the 10 Register Synchronization Scenarios

Table 2: Register Map Synchronization Scenarios

	Case	Description	Action
1	$a \notin (\text{dom } E \cup \text{dom } W)$	W(...) E(...) The abstract register is neither in the working rmap or the entry rmap.	Nil
2	$a \in \text{dom } W$ $\wedge$ $a \notin \text{dom } E$ $\wedge$ $W(a) \notin \text{rng } E$	W(a=>t1,...) E(...) The abstract register is in the working rmap, but not in the entry rmap. Furthermore the target register used in the working rmap is not in the range of the entry rmap.	Spill(W(a))
3	$a \in \text{dom } W$ $\wedge$ $a \notin \text{dom } E$ $\wedge$ $W(a) \in \text{rng } E$	W(a1=>t1,...) E(ax=>t1,...) The abstract register is in the working, but not in the entry rmap. However the target register used in the working rmap is in the range of the entry rmap.	Spill(W(a))
4	$a \notin \text{dom } W$ $\wedge$ $a \in \text{dom } E$ $\wedge$ $E(a) \notin \text{rng } W$	W(...) E(a1=>t1,...) The abstract register is in the entry rmap but not in the working rmap. Furthermore the target register used in the entry rmap is not in the range of the working rmap.	Fill(E(a), a)
5	$a \notin \text{dom } W$ $\wedge$ $a \in \text{dom } E$ $\wedge$ $E(a) \in \text{rng } W$	W(ax=>t1,...) E(a1=>t1,...) The abstract register is in the entry rmap but not in the working rmap. However the target register used in the entry rmap is in the range of the working rmap.	Reallocate(E(a)) Fill(E(a), a)
6	$a \in (\text{dom } W \cap \text{dom } E)$ $\wedge$ $W(a) \notin \text{rng } E$ $\wedge$ $E(a) \notin \text{rng } W$	W(a1=>t1,...) E(a1=>t2,...) The abstract register is in the working rmap and the entry rmap. However both use different target registers. Furthermore the target register used in the working rmap is not in the range of the entry rmap and the target register used in the entry rmap is not in the range of the working rmap.	Copy W(a) => E(a) FreeNoSpill(W(a))
7	$a \in (\text{dom } W \cap \text{dom } E)$ $\wedge$ $W(a) \notin \text{rng } E$ $\wedge$ $E(a) \in \text{rng } W$	W(a1=>t1,ax=>t2...) E(a1=>t2,...) The abstract register in the working rmap is in the entry rmap. However both use different target registers. The target register used in the working rmap is not in the range of the entry rmap, however the target register used in the entry rmap is in the range of the working rmap.	Spill(E(a)) Copy W(a) => E(a) FreeNoSpill(W(a))

Table 2: Register Map Synchronization Scenarios			
	Case	**Description**	**Action**
8	a ∈ (dom W ∩ dom E) ∧ W(a) ∈ rng E ∧ E(a) ∉ rng W	W(a1=>t1,...) E(a1=>t2,ax=>t1,...) The abstract register in the working rmap is in the entry rmap. However both use different target registers. The target register used in the entry rmap is not in the range of the working rmap, however the target register used in the working rmap is in the range of the entry rmap.	Copy W(a) => E(a) FreeNoSpill(W(a))
9	a ∈ (dom W ∩ dom E) ∧ W(a) ∈ rng E ∧ E(a) ∈ rng W ∧ W(a) ≠ E(a)	W(a1=>t1,ax=>t2,...) E(a1=>t2,ay=>t1,...) The abstract register in the working rmap is in the entry rmap. Both use different target registers. However, the target register used in the entry rmap is in the range of the working rmap, and the target register used in the working rmap is in the range of the entry rmap.	Spill(E(a)) Copy W(a) => E(a) FreeNoSpill(W(a))
10	a ∈ (dom W ∩ dom E) ∧ W(a) ∈ rng E ∧ E(a) ∈ rng W ∧ W(a) = E(a)	W(a1=>t1,...) E(a1=>t1,...) The abstract register in the working rmap is in the entry rmap. Furthermore they both map to the same target register.	Nil

The translator 19 performs two levels of register allocation within a group block, global and local (or temporary). Global register allocation is the definition of particular register mappings, before code generation, which persist across an entire group block (i.e., throughout all member blocks). Local register allocation consists of the register mappings created in the process of code generation. Global register allocation defines particular register allocation constraints which parameterize the code generation of member blocks, by constraining local register allocation.

Abstract registers that are globally allocated do not require synchronization on member block boundaries, because they are guaranteed to be allocated to the same respective target registers in every member block. This approach has the advantage that synchronization code (which compensates for differences in register mappings between blocks) is never required for globally allocated abstract registers on member block boundaries. The disadvantage of group block register mapping is that it hinders local register allocation because the globally allocated target registers are not immediately available for new mappings. To compensate, the number of global register mappings may be limited for a particular group block.

The number and selection of actual global register allocations is defined by a global register allocation policy. The global register allocation policy is configurable based on subject architecture, target architecture, and applications translated. The optimal number of globally allocated registers is derived empirically, and is a function of the number of target registers, the number of subject registers, the type of application being translated, and application usage patterns. The number is generally a fraction of the total number of target registers minus some small number to ensure that enough target registers remain for temporary values.

In cases where there are many subject registers but few target registers, such as the MIPS-X86 and PowerPC-X86 translators, the number of globally allocated registers is zero. This is because the X86 architecture has so few

target registers that using any fixed register allocation has been observed to produce worse target code than none at all.

In cases where there are many subject registers and many target registers, such as the X86-MIPS translator, the number of globally allocated registers (n) is three quarters the number of target registers (T). Hence:

$$\text{X86-MIPS:} \quad n = \tfrac{3}{4} * T$$

Even though the X86 architecture has few general purpose registers, it is treated as having many subject registers because many abstract registers are necessary to emulate the complex X86 processor state (including, e.g., condition code flags).

In cases where the number of subject registers and target registers is approximately the same, such as the MIPS-MIPS accelerator, most target registers are globally allocated with only a few reserved for temporary values. Hence:

$$\text{MIPS-MIPS:} \quad n = T - 3$$

In cases where the total number of subject registers in use across the entire group block (s) is less than or equal to the number of target registers (T), all subject registers are globally mapped. This means that the entire register map is constant across all member blocks. In the special case where ($s = T$), meaning that the number of target registers and active subject registers is equal, this means that there are no target registers left for

temporary calculations; in this case, temporary values are locally allocated to target registers that are globally allocated to subject registers that have no further uses within the same expression tree (such information is obtained through liveness analysis).

At the end of group block creation, code generation is performed for each member block, in the traversal order. During code generation, each member block's IR forest is (re)generated and the list of dead subject registers (contained in that block's liveness information) is used to the prune the IR forest prior to generating target code. As each member block is translated, its exit register map is propagated to the entry register maps of all successor member blocks (except those which have already been fixed). Because blocks are translated in traversal order, this has the effect of minimizing register map synchronization along hot paths, as well as making hot path translations contiguous in the target memory space. As with basic block translations, group member block translations are specialized on a set of entry conditions, namely the current working conditions when the group block was created.

Figure 7 provides an example of group block generation by the translator code 19 according to an illustrative embodiment. The example group block has five members ("A" to "E"), and initially one entry point ("Entry 1"; Entry 2 is generated later through aggregation, as discussed below) and three exit points("Exit 1," "Exit 2," and "Exit 3"). In this example, the trigger threshold for group block creation is an execution count of 45000, and the inclusion threshold for member blocks is an execution

count of 1000. The construction of this group block was triggered when block A's execution count (now 45074) reached the trigger threshold of 45000, at which point a search of the control flow graph was performed in order to identify the group block members. In this example, five blocks were found that exceeded the inclusion threshold of 1000. Once the member blocks are identified, an ordered depth first search (ordered by profiling metric) is performed such that hotter blocks and their successors are processed first; this produces a set of blocks with a critical path ordering.

At this stage global dead code elimination is performed. Each member block is analyzed for register uses and definitions (i.e., liveness analysis). This makes code generation more efficient in two ways. First, local register allocation can take into account which subject registers are live in the group block (i.e., which subject registers will be used in the current or successor member blocks), which helps to minimize the cost of spills; dead registers are spilled first, because they do not need to be restored. In addition, if liveness analysis shows that a particular subject register is defined, used, and then redefined (overwritten), the value can be thrown away any time after the last use (i.e., its target register can be freed). If liveness analysis shows that a particular subject register value is defined and then redefined without any intervening uses (unlikely, as this would mean that the subject compiler generated dead code), then the corresponding IR tree for that value can be thrown away, such that no target code is ever generated for it.

Global register allocation is next. The translator 19 assigns frequently accessed subject registers a fixed target register mapping which is constant across all member blocks. Globally allocated registers are non-spillable, meaning that those target registers are unavailable to local register allocation. A percentage of target registers must be kept for temporary subject register mappings when there are more subject registers than target registers. In special cases where the entire set of subject registers within the group block can fit into target registers, spills and fills are completely avoided. As illustrated in Figure 7, the translator plants code ("Pr1") to load these registers from the global register store 27 prior to entering the head of the group block ("A"); such code is referred to as prologue loads.

The group block is now ready for target code generation. During code generation, the translator 19 uses a working register map (the mapping between abstract registers and target registers) to keep track of register allocation. The value of the working register map at the beginning of each member block is recorded in that block's associated entry register map 40.

First the prologue block Pr1 is generated which loads the globally allocated abstract registers. At this point the working register map at the end of Pr1 is copied to the entry register map 40 of block A.

Block A is then translated, planting target code directly following the target code for Pr1. Control flow code is planted to handle the exit condition for Exit 1,

which consists of a dummy branch (to be patched later) to epilogue block Ep1 (to be planted later). At the end of block A, the working register map is copied to the entry register map 40 of block B. This fixing of B's entry register map 40 has two consequences: first, no synchronization is necessary on the path from A to B; second, entry to B from any other block (i.e., a member block of this group block or a member block of another group block using aggregation) requires synchronization of that block's exit register map with B's entry register map.

Block B is next on the critical path. Its target code is planted directly following block A, and code to handle the two successors, C and A, is then planted. The first successor, block C, has not yet had its entry register map 40 fixed, so the working register map is simply copied into C's entry register map. The second successor, block A, however, has previously had its entry register map 40 fixed and therefore the working register map at the end of block B and the entry register map 40 of block A may differ. Any difference in the register maps requires some synchronization ("B-A") along the path from block B to block A in order to bring the working register map into line with the entry register map 40. This synchronization takes the form of register spills, fills, and swaps and is detailed in the ten register map synchronization scenarios above.

Block C is now translated and target code is planted directly following block C. Blocks D and E are likewise translated and planted contiguously. The path from E to A again requires register map synchronization, from E's exit

register map (i.e., the working register map at the end of E's translation) to A's entry register map 40, which is planted in block "E-A."

Prior to exiting the group block and returning control to the translator 19, the globally allocated registers must be synchronized to the global register store; this code is referred to as epilogue saves. After the member blocks have been translated, code generation plants epilogue blocks for all exit points (Ep1, Ep2, and Ep3), and fixes the branch targets throughout the member blocks.

In embodiments that use both isoblocks and group blocks, the control flow graph traversal is made in terms of unique subject blocks (i.e., a particular basic block in the subject code) rather than isoblocks of that block. As such, isoblocks are transparent to group block creation. No special distinction is made with respect to subject blocks that have one translation or multiple translations.

In the illustrative embodiment, both the group block and isoblock optimizations may be advantageously employed. However, the fact that the isoblock mechanism may create different basic block translations for the same subject code sequence complicates the process of deciding which blocks to include in the group block, since the blocks to be included may not exist until the group block is formed. The information collected using the unspecialized blocks that existed prior to the optimization must be adapted before being used in the selection and layout process.

The illustrative embodiment further employs a technique for accommodating features of nested loops in group block generation. Group blocks are originally created with only one entry point, namely the start of the trigger block. Nested loops in a program cause the inner loop to become hot first, creating a group block representing the inner loop. Later, the outer loop becomes hot, creating a new group block that includes all the blocks of the inner loop as well as the outer loop. If the group block generation algorithm does not take account of the work done for the inner loop, but instead re-does all of that work, then programs that contain deeply nested loops will progressively generate larger and larger group blocks, requiring more storage and more work on each group block generation. In addition, the older (inner) group blocks may become unreachable and therefore provide little or no benefit.

According to the illustrative embodiment, group block aggregation is used to enable a previously built group block to be combined with additional optimized blocks. During the phase in which blocks are selected for inclusion in a new group block, those candidates which are already included in a previous group block are identified. Rather than planting target code for these blocks, aggregation is performed, whereby the translator 19 creates a link to the appropriate location in the existing group block. Because these links may jump to the middle of the existing group block, the working register map corresponding to that location must be enforced; accordingly, the code planted for the link includes register map synchronization code as required.

The entry register map 40 stored in the basic block data structure 30 supports group block aggregation. Aggregation allows other translated code to jump into the middle of a group block, using the beginning of the member block as an entry point. Such entry points require that the current working register map be synchronized to the member block's entry register map 40, which the translator 19 implements by planting synchronization code (i.e., spills and fills) between the exit point of the predecessor and the entry point of the member block.

In one embodiment, some member blocks' register maps are selectively deleted to conserve resources. Initially, the entry register maps of all member blocks in a group are stored indefinitely, to facilitate entry into the group block (from an aggregate group block) at the beginning of any member block. As group blocks become large, some register maps may be deleted to conserve memory. If this happens, aggregation effectively divides the group block into regions, some of which (i.e., member blocks whose register maps have been deleted) are inaccessible to aggregate entry. Different policies are used to determine which register maps to store. One policy is to store all register maps of all member blocks (i.e., never delete). An alternative policy is to store register maps only for the hottest member blocks. An alternative policy is to store register maps only for member blocks that are the destinations of backward branches (i.e., the start of a loop).

In another embodiment, the data associated with each group member block includes a recorded register map for every subject instruction location. This allows other

translated code to jump into the middle of a group block at any point, not just the beginning of a member block, as, in some cases, a group member block may contain undetected entry points when the group block is formed. This technique consumes large amounts of memory, and is therefore only appropriate when memory conservation is not a concern.

Group blocking provides a mechanism for identifying frequently executed blocks or sets of blocks and performing additional optimizations on them. Because more computationally expensive optimizations are applied to group blocks, their formation is preferably confined to basic blocks which are known to execute frequently. In the case of group blocks, the extra computation is justified by frequent execution; contiguous blocks which are executed frequently are referred to as a "hot path."

Embodiments may be configured wherein multiple levels of frequency and optimization are used, such that the translator 19 detects multiple tiers of frequently executed basic blocks, and increasingly complex optimizations are applied. Alternately, and as described above only two levels of optimization are used: basic optimizations are applied to all basic blocks, and a single set of further optimizations are applied to group blocks using the group block creation mechanism described above.

Figure 8 illustrates the steps performed by the translator at run-time, between executions of translated code. When a first basic block (BB_{N-1}) finishes execution 1201, it returns control to the translator 1202. The

translator increments the profiling metric of the first basic block 1203. The translator then queries the basic block cache 1205 for previously translated isoblocks of the current basic block (BB_N, which is BB_{N-1}'s successor), using the subject address returned by the first basic block's execution. If the successor block has already been translated, the basic block cache will return one or more basic block data structures. The translator then compares the successor's profiling metric to the group block trigger threshold 1207 (this may involve aggregating the profiling metrics of multiple isoblocks). If the threshold is not met, the translator then checks if any isoblocks returned by the basic block cache are compatible with the working conditions (i.e., isoblocks with entry conditions identical to the exit conditions of BB_{N-1}). If a compatible isoblock is found, that translation is executed 1211.

If the successor profiling metric exceeds the group block trigger threshold, then a new group block is created 1213 and executed 1211, as discussed above, even if a compatible isoblock exists.

If the basic block does not return any isoblocks, or none of the isoblocks returned are compatible, then the current block is translated 1217 into an isoblock specialized on the current working conditions, as discussed above. At the end of decoding BB_N, if the successor of BB_N (BB_{N+1}) is statically determinable 1219, then an extended basic is created 1215. If an extended basic block is created, then BB_{N+1} is translated 1217, and so forth. When translation is complete, the new isoblock

is stored in the basic block cache 1221 and then executed 1211.

Partitions

In a preferred embodiment, the translator 19 employs a partitioning technique to divide the subject code space into regions, referred to hereafter as partitions. Each partition includes a distinct set of translator data structures (e.g., basic blocks) and target code. A partition represents a range of subject code 17 addresses which does not overlap with any other partition, meaning that no translated block spans between two partitions. As such, all translations of a particular subject code address are present in only a single partition.

The partitioning technique utilized by the translator 19 divides the translator 19's representation of control flow into non-overlapping regions of subject memory. In some cases, these partitions correspond to the different libraries and object files that make up a subject program, such that partitions are created and replaced as libraries are mapped into subject memory and later discarded. The partitioning technique is particularly utilized by the translator 19 when translating subject programs that modify their own subject code 17, hereinafter referred to as "self-modifying". By dividing subject code 17 and its corresponding translated target code 21 into partitions, one partition can be discarded when its associated code is modified without affecting the valuable information (e.g., translations) built up in other partitions. Without the use of such partitions, all existing translations, even

those not affected by the code modification, would need to be discarded in response to every code modification.

In a preferred embodiment, the translator 19 maintains segregated data structures, such that translation data structures are organized by partitions. This segregation allows for the numerous block translations (and affiliated data structures) that are associated with a single partition to be discarded and freed at once, rather than traversing the underlying representations of translated code to search for affected translations and deleting them one at a time. For example, in one embodiment the translator 19 maintains a separate basic block cache 23 for each partition.

When subject code 17 modifies other subject code, the "modification event" is detected by the translator 19. A modification event is defined to correspond to a particular subject address range (the range of subject code that is overwritten or deleted by the modification). The translator 19 must first detect when and where subject code is self-modified (i.e., which subject code performs the modification, and which subject code is modified). Modification events are any events when subject code 17 is modified by subject code 17. Self-modifying code does not include cases in which subject code 17 modifies subject data. Self-modification of subject code 17 can take many forms, including but not limited to: (i) mapping a file into memory (e.g., mmap() system calls); (ii) removing a file from memory (e.g., munmap() system calls); and (iii) making a memory region executable (e.g., changing its permissions using the mprotect() system call).

Alternatively, in cases where subject code modifications are made using a well known system call, such as mmap(), munmap() or mprotect(), the translator 19 detects all calls to that system call in the subject code. The translator 19 can also detect subject code 17 modifications by other mechanisms. For example, many subject processors require that, prior to executing code that has been modified, the subject program must first flush the processor's instruction cache (I-cache). In particular, the PowerPC architecture has a special instruction for this purpose, "ICBI" (Instruction Cache Block Invalidate), while other architectures may use a special system call for the same purpose. On architectures with such a cache flush requirement, the translator 19 may detect modification events by detecting instances of the special instruction (or special system call).

Alternatively, the translator 19 can use features of the target operating system 20 to monitor all writes to the target memory regions which correspond to subject code, in order to detect modification events. On certain systems, the system call mprotect() allows the translator 19 to set particular regions of memory "read-only". Other systems may utilize equivalent functions to that of the mprotect() system call to define a particular area of memory as read-only. Any attempt to write into read-only regions triggers a signal which is detected by the translator 19. The signal notifies the translator 19 that a subject code modification is taking place, and the translator 19 uses the signal context to determine which subject address is being overwritten. After detecting the modification and identifying the scope of the modification

(i.e., the subject addresses affected), the translator 19 then allows the write to proceed normally.

Alternatively, the translator 19 can generate a special target code sequence, for each translated memory write operation in the subject code, which checks if the write address corresponds to subject code rather than subject data, in order to detect modification events.

After a modification event has been detected, the modification results in the creation of a new partition whose subject address range is the same as the modified range. In addition, if any existing partitions had address ranges that intersected (overlapped) the modified range, all such intersecting partitions are destroyed and re-created, except in certain situations where additional partition optimizations are applied as described below. For each intersecting partition that is destroyed a new remainder partition is created for the remaining range, meaning the original range of the intersecting partition minus the intersection. When a partition is destroyed, all existing target code 21 translations associated with that partition are discarded. As such, a new partition is, at first, completely empty of translated target code 21, meaning that any subject code 17 subsequently encountered from that partition must be translated from scratch.

Figure 9 illustrates the creation of a new partition, from a set of existing partitions and a modification event which overlaps some of those partitions. Prior to the modification, four partitions A, B, C and D exist. The subject code then modifies a range of subject code

addresses extending from point 100 in memory to point 102. The range of the modification 104 intersects with partitions A and C and totally encompasses partition B. Partition M is created for the entire modified range 104. Partitions A and C are destroyed, and partition A' and C' are created, respectively, for the remaining portions of A and C which do not intersect the modification range 104. Partition B is destroyed and its range is completely subsumed by the new partition M. Partition D, whose range does not overlap the modification range 104, remains unaffected.

One optimization technique that can be applied to partitioning in order to improve performance of the translator 19 is lazy partition allocation. In order to save memory, the full data structures of a partition are not allocated at the time a partition is created. Instead, an inactive partition is initially created which reserves the partition's subject address range, but the translator 19 does not initially allocate any of the underlying translation data structures and memory. In this manner, the initial inactive partitions are essentially empty skeleton partitions that merely reserve a particular subject address range. When any subject code 17 within the partition is actually translated, the inactive partition becomes a live partition and the partition data structures are initialized. Accordingly, modification events which correspond to data segments of the subject program (e.g., files which are mmap()ed as data, and which the subject program never executes as code) cause a new inactive partition to be created, but none of the underlying translation data structures and memory regions are allocated.

In a preferred embodiment, the translator 19 allocates large regions of memory for new partitions to store target code 21 translations. Target code 21 is preferably allocated to be contiguous so as to avoid unnecessary fragmentation. The translator 19 is more likely to be able to make target code 21 contiguous if it has a large memory region to use. Partitions which are falsely detected (i.e., modification events in subject memory ranges which do not correspond to executable subject code) can sometimes consume valuable memory resources for no purpose. By only allocating translation data structures and memory when subject code 17 within a partition is actually translated (and executed), lazy partition allocation allows the translator 19 to avoid the negative impact of false positives from the modification event detection mechanism associated with partitioning.

In certain situations, the regions of the subject memory space which the translator 19 identifies as being modified by a modification event may never actually be translated into target code. This may be because the subject memory region corresponds to data, or because the subject memory region corresponds to code which is never executed. To avoid the overhead of wasted memory resources, an inactive or skeleton partition is created initially, where the partition's full data structures are only realized when translation occurs within the partition's range of subject addresses. Because inactive partitions contain no translations of subject code, they can be created, deleted, or resized without the memory and performance overhead associated with live partitions.

In an alternative embodiment of the translator 19 which also employs interpreter functionality, the translator 19 may invoke interpreter functions to interpret subject code 17 rather than translating it, specifically for the purpose of delaying translation and thereby avoiding the allocation of partitions associated with that subject code 17. A translation method and apparatus which facilitates such interpreter functionality is described in co-pending UK Patent Applications Serial No. 03 09056.0 and 03 15164.4, both entitled "Block Translation Optimizations for Program Code Conversion", the disclosure of which is incorporated herein by reference.

In another preferred embodiment of the translator 19, an optimization technique that can be applied to partitioning in order to improve performance of the translator 19 is reducible partitions, where the address range of a partition is reduced rather than deleting the partition, to a range which includes the range of addresses actually used in the partition. Initially in response to a modification event, a new partition is created with a starting address and an ending address whose values are defined by the scope of the modification. However, in operation, the actual range of subject addresses that are translated may be narrower than the defined range of the partition. The range encompassing all subject addresses within a partition that have actually been translated or which are flagged for future translation is referred to as the "active range" or used range of a partition. The range of subject addresses in a partition that have not been translated is referred to as the "inactive range." In order to optimize the

performance of the translator 19, the size of a partition may be reduced, from its initially defined range of addresses to a lesser range which includes its active range of addresses by eliminating at least a portion of the inactive range of addresses.

While the concept of reducible partitions may be implemented in any number of ways, in one preferred embodiment, the translator 19 maintains the active range or translated range of addresses in each partition. When a partition is created, its active range of addresses is initially empty. During translation, as each subject instruction is translated, its subject address ("the translated address") is compared to the active range of the current partition. If the translated address is outside the current active range, the active range is expanded to include the translated address. As such, the active range will grow as translation of the subject code progresses.

When the translator 19 detects a modification event, the translator 19 initially determines if the range of the modification overlaps the initially defined range of an existing partition. If an overlap exists, the translator 19 then determines whether the modification range actually overlaps the partition's active range of subject addresses. If the modification range does not overlap the active range, then the translator 19 can simply resize the partition's initially defined range without deleting the translation data structures within the partition. The reason is that, while the initially defined partition range represents the potential scope of the partition, as defined by the prior modification event which created the

partition, the active range represents the actual range of subject addresses translated. Modifications which do not overlap the active range therefore do not invalidate any of the translations within the partition, because the translator 19 knows that the translated subject addresses are distinct from, and therefore unaffected by, the modified subject addresses. By resizing the partition to a range which includes the active range, the translations stored within the partition can be kept in that partition and need not be deleted. On the other hand, if the modification range does overlap the partition's active range of addresses, then the translator 19 must delete the entire partition as described above.

Referring now to Figure 10, an illustration of the reducible partition optimization technique is provided. The partitions illustrated in Figure 10 are substantially the same as those shown in Figure 9, except that partition A includes a active range of subject addresses 106. Prior to the modification, four partitions A, B, C and D exist. The subject code then modifies a range of subject code addresses extending from point 100 in memory to point 102. The range of the modification 104 intersects with the initially defined partition range for partition A, but the range of modification does not intersect with the active range 106 of partition A. After partition M is created for the modified range 104, the range of partition A can be reduced without affecting the translations in partition A. Contrarily, it can be seen in this example that the range of modification 104 intersects with the active range 108 of partition C. Thus, the range of partition C can not simply be reduced and partition C must be destroyed, whereupon new partition C' is created for the remaining

portion of partition C which is not intersecting with the range of modification 104. Partition B is destroyed and its range is completely subsumed by the new partition M. Partition D remains unaffected.

The translator 19 can detect most modification events at decode-time, but the actual subject addresses to be modified may not be known until the translated code 21 is executed. Thus, when the translator 19 detects during decoding either a subject code 17 modification or an indication that a modification has occurred, the translator 19 ends the translation of the current block, and inserts a notification immediately after the current block to notify the translator of the modification event.

In one embodiment of the translator 19, the partitioning technique is implemented in the translator 19 with the aid of special blocks. Special blocks are blocks which, while they may correspond to a particular subject address do not represent translations of subject code. In contrast, a translation block as described above represents a translation of a particular subject code sequence beginning with a particular starting subject address. Instead, special blocks contain special actions of the translator 19 which are inserted into the stream of target code during translation, before the blocks are actually executed. In effect, special blocks are lightweight translator actions, which can be planted at particular points in the target code control flow without requiring an expensive context switch out of target code back to the translator loop.

Special blocks contain pseudo-target code rather than normal target code. Normal target code represents a translation of some subject code sequence. Pseudo-target code consists of artificial (i.e., not translated) target code sequences, which is either written directly in target code or written in a high-level programming language (e.g., C++) and compiled to have the same calling conventions as target code generated by the translator. Examples of special blocks include notifyBlocks and border guard blocks, which are discussed herein.

In one embodiment, the translator inserts a special block called a "notifyBlock" into the control flow of the translated program at a point immediately following the block containing the self-modifying subject code. The notifyBlock is a "special" translation structure, because unlike a translation block it does not represent the translation of any particular subject code, but rather it represents the modification event. The subject code which actually performs the modification is translated as part of the current block. The notifyBlock is inserted as the successor of the current block. Thus, immediately after the modification is performed in the translated current block, the subsequent notifyBlock notifies the translator to perform the appropriate actions in response to that modification (i.e., partition adjustments). The subject code 17 that follows the modification event is translated in a new block, which becomes the successor of the notifyBlock.

When the notifyBlock is reached in the target code 21 at run-time, the translator 19 is notified of the subject address range of the modified code. The translator 19

uses this information to create a new partition, which may also alter or destroy existing partitions.

The block containing the self-modifying subject code (the predecessor of the notifyBlock) ends immediately after identifying a modification event because the modification may invalidate the current partition. The modified subject code might be the next subject instruction, which would require the translator to translate and execute the newly modified version of that instruction rather than executing an existing translation of the old version. For example, on the PowerPC architecture, a block must end after a cache flush instruction. Thus, when a modification event is detected, the current block of translation is ended and a notifyBlock is inserted as the successor.

The notifyBlock also copies the current subject address and compatibility list from its predecessor block to its successor block. After the notification occurs, translation resumes in the successor block at the next subject instruction. If the current partition is destroyed as a result of the operation, the notifyBlock also handles the safe transition into the new partition which replaces it.

Because translation blocks must end at partition boundaries, partitions can have a negative impact on the optimizations described above. Sections of code which would otherwise be translated together are translated as separate blocks due to the partitions formed, reducing the scope of optimization and increasing the number of returns to the translator loop. For example, the extended block

and group block optimizations are limited by partition boundaries. For this reason, one optimization that can be applied to partitioning in order to improve performance of the translator is, for subject programs that are known not to modify their own subject code, aggregating all subject memory into a single partition such that all subject code is contained in one partition.

The scope of a partition must be properly defined because partition scope impacts the performance of the translator. If partitions are too small, control flow must constantly pass through border guards (described below). If partitions are too large, a slight code modification will unnecessarily invalidate larger portions of translated code. A proper balance of such constraints should be considered when selecting partition size as it relates to the performance of the translator. Optimizations can be applied if it is determined that the particular partitions being generated have a negative impact on performance. By default, partition size is determined by the modification event detection mechanism. For example, if the translator 19 detects an mmap() of a run-time library, a new partition is created that encompasses the entire library.

One optimization technique that can be applied to partitioning in order to improve performance of the translator 19 is the aggregation of partitions. For example, the PowerPC ICBI cache flush instruction invalidates one page of executable memory. However, if multiple pages of subject code 17 are modified at once, the subject program may contain several consecutive ICBI instructions which invalidate contiguous pages of memory.

Under the partitioning mechanism described above, this would result in multiple, contiguous, page-wide partitions. As such, one optimization of the partition mechanism is to detect consecutive cache flush instructions and coalesce the modified ranges into one "aggregated" partition.

In one embodiment of the aggregation optimization, when a new partition is created, the translator 19 checks if there is an existing partition (i.e., not a remainder partition created as a byproduct of the new partition's creation) that is adjacent to and precedes the new partition in the subject code. If so, the range of the preceding existing partition can be expanded to include the range of the new partition, effectively aggregating the two partitions.

The translator 19 may further aggregate those regions of subject code 17 which are known not to be modified, when such regions are contiguous. For example, in the situation where an entire subject program is known not to modify its own subject code, the translator 19 uses a single partition for all of the subject code 17. By further example, on the PowerPC architecture, all shared system libraries are located in the subject address range 0x90000000 - 0xA0000000 (the PowerPC "shared library region"). Most applications never modify the system libraries, such that the translator 19 may aggregate the entire PowerPC shared library region into one partition.

In one embodiment of the partitioning technique, the translator inserts an additional level of indirection for control flow that travels between partitions. The

translator 19 uses special placeholders, referred to herein as border guards, for blocks that touch partition boundaries, so that predecessor blocks can check if previously translated successor blocks still exist. With this approach, predecessor blocks can be efficiently notified when their successors are deleted. The translator 19 inserts a pair of border guard blocks at every point where the subject program's control flow crosses a partition boundary, otherwise referred to as border crossings. The pair of border guard blocks includes an exit border guard block and an entry border guard block. An exit border guard block is added after, and in the same partition as, the predecessor block. An entry border guard block is added before, and in the same partition as, the successor block.

For translated blocks within the same partition, control flow passes directly from the predecessor block to the successor block. For blocks in different partitions (i.e., border crossings), control flows from the predecessor block to an exit border guard, then to an entry border guard, and then to the successor block.

Entry border guards serve as placeholders for the translator 19, where entry border guards store references to their exit border guard counterparts. However, entry border guards perform minimal actions when "executed" (i.e., when control flow passes through them). The execution of an exit border guard verifies that a previously translated successor still exists. The data structure of each exit border guard also contains a reference to its entry border guard counterpart. Border guard blocks serve a bookkeeping function by providing an

explicit representation of partition crossings. When the translator 19 deletes a particular partition, it traverses every entry border guard within that partition to find the corresponding exit guards. As the partition is deleted, the counterpart reference of each such exit border guard is set to null. In some embodiments, the translator 19 stores all of the entry border guards for a given partition together, such that they can be traversed efficiently when the partition is deleted. For example, in one embodiment, the translator 19 maintains an entry border guard list for each partition, wherein such list is updated whenever entry border guards are created or deleted.

Accordingly, when an exit border guard is executed, it verifies that its successor still exists by simply checking its own counterpart reference. If the reference is defined, then a valid translated successor exists and control passes to the respective entry border guard (and therefore to the successor partition). If the reference is undefined, then the exit border guard is unaware of a valid translated successor block, meaning that either there was one and it was deleted or this exit border guard has never been executed and therefore the successor has never been determined. If the counterpart reference is undefined, a new successor block is obtained by either checking the basic block cache of the respective partition or translating the successor block. From the exit border guard's perspective, a successor block that was translated and subsequently deleted is indistinguishable from a successor block that has never been translated.

The explicit representation of partition border crossings achieved by the border guard pair makes the process of partition deletion more efficient. To delete a partition, the translator 19 must identify and nullify all references to deleted blocks (i.e., previously translated blocks within the deleted partition). Absent some bookkeeping mechanism, such as border guards, the translator 19 would need to traverse numerous translated blocks to identify which blocks had successors in the deleted partition, in order to remove the references to those discarded successors. In addition to making the process of deleting a partition more efficient, border guard blocks facilitate thread-safe border crossings, as discussed below.

To delete a partition in the presence of border guards, the translator 19: (i) voids the "successor" links of all exit guards that point to the partition, (ii) notifies all successor partitions that their corresponding entry guards can be discarded, and (iii) deletes all translation structures and target code 21 belonging to the deleted partition. In the first step, the translator 19 traverses all of the entry border guards in the partition, and resets their foreign exit border guard counterparts, effectively notifying all predecessor partitions that the deleted partition is void. In the second step of partition deletion, all of the exit border guards in the partition are traversed to notify the successor partitions that their corresponding entry guards can be discarded. In one embodiment, the second step is performed by traversing each exit border guard of the deleted partition. In another embodiment, all of the entry guards in a partition are indexed by the predecessor partition

which contains the corresponding exit guard. In this case, the deleted partition need only notify each successor partition once, and each successor partition can then identify the corresponding entry points to be deleted.

Multithreaded Programs in Partitioning

Multithreaded subject programs present a difficulty to the translator 19 when utilizing partitioning, namely the deletion of partitions must be performed in a thread-safe manner. Specifically, while a given partition is being deleted, threads must be prevented from entering that partition. This guarantees that control always flows into valid translations, and never into a deleted or invalid block. After a partition has been deleted and recreated, threads may then be allowed to enter the (newly emptied) partition.

In a preferred embodiment, the translator 19 uses a single global mutex ("the global partition lock") to serialize particular partition operations, including: (i) control flow jumps between partitions (i.e., border crossings); and (ii) deletion, and where applicable re-creation, of partitions. Thus, an exit border guard must acquire the global partition lock before control passes to its corresponding entry border guard. Likewise, a notifyBlock must acquire the global partition lock before destroying a partition. For modification events that require the deletion and subsequent re-creation of one or more partitions, all such deletions and re-creations are performed atomically under the protection of the global partition lock, meaning that all such operations are

performed in sequence without releasing the lock. The translator 19 maintains a partition identifier for each thread, which is changed when control passes between partitions. This change is made by a border guard pair while holding the global partition lock.

Referring now to FIG. 11, the control flow of a translated program across partition boundaries is illustrated, both with and without crossing a partition boundary. The top half of FIG. 11 shows the control flow of a particular subject program in the translator 19 when partitioning is not being employed or when control flow remains within a single partition. Control passes from subject block A 201 to subject block B 203, then to subject block C 205, then to subject block D 207. In terms of the program subject code, block B 203 is block A's 201 successor, block C 205 is block B's successor, and block D 207 is block C's successor.

Alternatively, assuming a partition boundary 209 existed between subject bock B and subject block C, control would necessarily flow through a border guard pair when crossing the partition boundary 209, as illustrated in the lower portion of FIG. 11. Blocks A 201 and B 203 are both in partition 215, while blocks C 205 and D 207 are both in partition 217. In the translated subject program, control passes directly from block A 201 to block B 203, because blocks A 201 and B 203 are in the same partition 215. The transfer of control from block B 203 to block C 205, however, crosses a partition boundary. As such, control passes from block B 203 to an exit border guard block 211. As described above, the exit border guard 211 checks that its corresponding entry border guard

213 has not been deleted. In a multithreaded program (discussed below), the exit border guard also acquires the global partition lock, so that the partition boundary can be crossed in a thread-safe manner. Control then passes from the exit border guard block 211 to its corresponding entry border guard block 213. Execution has now passed into partition 217. In a multithreaded program (discussed below), the entry border guard changes the partition identifier of the current thread to reflect the fact that the thread is now in partition 217 and the global partition lock is then released. Once inside partition 217, control passes from the entry border guard 213 to block C 205, and then directly from block C 205 to block D 207.

Memory Management

Memory management is critical consideration in the various embodiments of the dynamic binary translator 19. The memory demands of translating a program from one architecture to another are high. The memory demands become even higher when optimizations such as isoblocks, extended blocks, and group blocks are introduced, as each of these optimizations creates the possibility that one sequence of subject code 17 may be represented by multiple translations. A modification event requires deletion of a partition, and therefore the deletion of all translation data (i.e., block structures and target code) therein.

In one embodiment, the translator 19 provides its own memory management subsystem which mirrors the subject code partitions. Partitions are intended to group all translator data structures by the subject code 17 regions

to which they correspond. Portions of subject code 17 that are likely to be invalidated together form a partition: all related translator data structures therefore reside in the same well-defined region of target memory. If a partition is invalidated, all of the translation data (translator structures and target code) associated with that partition can be freed en masse, avoiding the need to free every structure individually.

In this specific embodiment, the translator 19 performs all memory allocation through memorySource objects. Each partition has one corresponding memorySource. MemorySources obtain memory from the operating system through conventional means such as mmap() system calls, but they do so in bulk. Other translator code (including pseudo-target code) obtains memory as needed (i.e., in smaller quantities), but from per-partition memorySources rather than directly from the operating system. This improves the performance of the translator 19 by reducing the number and frequency of underlying memory-related system calls. MemorySources also provide a function for flushing the entire contents of the memorySource at once. The memorySource can implement a flush by actually freeing the underlying memory or by simply discarding all of the "allocations" it has made from that memory (i.e., wiping the slate clean while retaining the underlying memory).

The memory subsystem of the translator 19 also simplifies the process of deleting a partition. To delete a partition, the translator 19 must: (i) void the "successor" links of all predecessor partitions' exit guards; (ii) notify all successor partitions that their

corresponding entry guards can be discarded; and (iii) delete all translation structures and target code belonging to the partition. Without the memory subsystem, the third step requires that the translator walk the per-partition collections of translation structures and target code to free each structure individually. With the memory subsystem, all of the partition's structures can be freed at once by simply flushing the memorySource.

Shared Cache

In another preferred embodiment, the translator 19 includes a shared code cache, which allows the target code 21 and translation structures corresponding to a particular subject program to be shared between different executions or runs of the translator 19. The shared code cache is facilitated by a dedicated code cache server process, which interacts with translators 19 at the beginning and end of their executions.

In one embodiment, a code cache consists of one or more memorySource objects and the regions of memory that they own (i.e., partitions), which are stored by the cache server as files. For example, in one embodiment of PowerPC translators (which translate from the PowerPC subject architecture to some target architecture), the partition representing the PowerPC shared library region (subject addresses 0x90000000 - 0xA0000000) is cached as one file in the shared code cache.

Each code cache contains target code and translation structures specific to both a particular subject program and to a particular compilation or build of the translator

itself (data structures from different translator builds may be binary-incompatible). To verify that an existing code cache is compatible with a particular run of the translator, various metrics are compared; any metric which yields non-identical values indicates binary incompatibility and renders the code cache incompatible. The code cache binary compatibility metrics include the date and time that the translator was built.

When translation of a subject program begins, the translator 19 checks the code cache server for a compatible code cache. If a compatible cache is found, the translator 19 loads the cache, which consists of target code 21 and translation structures. A cache potentially contains all of the translated code 21 created over the course of a translated execution of the subject program, including optimized target code such as group blocks. This allows a later translator execution to piggyback on the efforts of earlier executions; large sections of subject code may have already been translated (reducing startup time and translation cost) and possibly optimized.

The translator 19 loads a code cache file by mapping it into memory as a shared object, similar to how a shared library is loaded. The cache file is preferably shared using a copy-on-write policy. Under a copy-on-write policy, the cache file is initially shared across all running translator processes. Copy-on-write means that when a particular translator execution modifies the cached structures in any way (e.g., incrementing a block's execution count) the cache becomes exclusive to that

particular execution and the memory region is no longer shared across multiple processes.

When a translator execution completes (i.e., the translated program terminates), the cache server compares that execution's code cache to the cache stored by the server. If the current execution's code cache is better (i.e., has more block translations) than the stored version, the server stores the translator's cache for future use. As such, the quality of the code caches stored in the server improve over time.

Although a few preferred embodiments have been shown and described, it will be appreciated by those skilled in the art that various changes and modifications might be made without departing from the scope of the invention, as defined in the appended claims.

Attention is directed to all papers and documents which are filed concurrently with or previous to this specification in connection with this application and which are open to public inspection with this specification, and the contents of all such papers and documents are incorporated herein by reference.

All of the features disclosed in this specification (including any accompanying claims, abstract and drawings), and/or all of the steps of any method or process so disclosed, may be combined in any combination, except combinations where at least some of such features and/or steps are mutually exclusive.

Each feature disclosed in this specification (including any accompanying claims, abstract and drawings) may be replaced by alternative features serving the same, equivalent or similar purpose, unless expressly stated otherwise. Thus, unless expressly stated otherwise, each feature disclosed is one example only of a generic series of equivalent or similar features.

The invention is not restricted to the details of the foregoing embodiment(s). The invention extends to any novel one, or any novel combination, of the features disclosed in this specification (including any accompanying claims, abstract and drawings), or to any novel one, or any novel combination, of the steps of any method or process so disclosed.

Claims

1. A method of partitioning code during a translation of subject code into translated target code to account for self-modifying subject code, comprising:

identifying self-modifying code events in said subject code; and

dividing a region of memory containing said subject code into at least one partition when identifying a self-modifying code event, wherein each partition includes a range of subject code addresses in said memory which are affected by a respective self-modifying code event.

2. The method of claim 1, wherein each partition further includes translated target code corresponding to subject code contained in that partition.

3. The method of claim 1 or 2, wherein each said partition represents a region of memory that does not overlap with regions of memory corresponding to other partitions.

4. The method of any preceding claim, wherein a self-modifying code event modifies a respective range of subject code addresses, said method further comprising:

modifying partitions existing in said memory that possess subject code addresses which are affected by said self-modifying code event.

5. The method of claim 4, wherein said partition modifying step comprises:

creating a new partition to include modified subject code corresponding to the self-modifying code event; and

for existing partitions having ranges of subject code addresses which overlap with the subject code addresses of the newly created partition, modifying said existing partitions to delete the subject code addresses from said existing partitions that overlap with the subject code addresses of the newly created partition such that the partitions no longer overlap.

6. The method of claim 5, wherein each partition further includes translated target code corresponding to subject code contained in that partition, said method further comprising:

deleting translated target code associated with partitions that have been modified in response to the self-modifying code event; and

translating new target code for the subject code contained in the modified partitions.

7. The method of claim 5 or 6, further comprising adding translated target code to a partition as corresponding subject code in that partition is translated.

8. The method of claim 7, wherein each partition includes a particular range of subject code addresses that

have been translated, such that the particular range of subject code addresses having been translated comprises an active range within the partition, said method further comprising:

determining whether the subject code addresses of said newly created partition overlap with any subject code addresses in said active range of any partition; and

for existing partitions having an active range that overlaps with the subject code addresses of said newly created partition, deleting translated target code associated with partitions that have been modified in response to the self-modifying code event, and translating new target code for the subject code contained in the modified partitions.

9. The method of claim 8, wherein each partition includes a range of subject code addresses that have not been translated referred to as an inactive range within the partition, said method further comprising:

for existing partitions having an active range which does not overlap with the subject code addresses of said newly created partition but having an inactive range that does overlap with the subject code addresses of said newly created partition, modifying said existing partitions to delete the subject code addresses from said inactive ranges in said existing partitions that overlap with the subject code addresses of the newly created partition such that the partitions no longer overlap, and leaving the translated target code associated with active ranges in said existing partitions unchanged.

10. The method of any of claims 4 to 9, further comprising:

identifying partitions that are adjacent to one another in memory having characteristics that allow them to be combined; and

aggregating said adjacent partitions into a single, combined partition.

11. The method of any preceding claim, wherein said self-modifying code event is identified during decoding of the subject code, said method further comprising inserting a special translation structure into a control flow of the translated target code as a representation of the identified self-modifying code event.

12. The method of claim 11, in response to encountering said special translation structure during execution of the translated target code, said method further comprising:

identifying the range of subject code addresses affected by the self-modifying code event, and creating the partition in memory using this identified range of subject code addresses.

13. The method of any preceding claim, wherein each partition includes a pair of border guards which facilitate control flow passing between partitions, wherein said pair of border guards includes an entry border guard and an exit border guard, such that each exit

border guard contains a specific reference to a counterpart entry border guard in a succeeding partition to be executed next.

14. The method of claim 13, when encountering an exit border guard during execution of a current partition, said method further comprising verifying that a counterpart entry border guard exists in a successive partition before passing control from the current partition to the successive partition.

15. The method of claim 13 or 14, wherein a set of border guards exists containing entry border guards and exit border guards for all partitions, said method further comprising modifying said set of border guards whenever a new partition is created in response a self-modifying code event.

16. The method of any of claims 4 to 15, wherein when subject code defines a multi-threaded program, said method further comprising preventing other threads from entering a partition while the partition is being modified by another thread.

17. The method of any of claims 4 to 16, wherein each partition further includes translated target code corresponding to subject code contained in that partition, wherein each partition includes a pair of border guards which facilitate control flow passing between partitions, wherein said pair of border guards includes an entry border guard and an exit border guard, such that each exit border guard contains a specific reference to a

counterpart entry border guard in a succeeding partition to be executed next, said method further comprising:

providing a memory management subsystem having regions which mirror the subject code partitions, wherein said memory management subsystem stores target code and border guard pairs associated with a partition along with its corresponding subject code; and

deleting an entire region of said memory management subsystem that corresponds to a specific partition whenever that specific partition is modified.

18. A computer readable storage medium having translator software resident thereon in the form of computer readable code executable by a computer for performing a method of partitioning code during a translation of subject code into translated target code to account for self-modifying subject code, said method comprising:

identifying self-modifying code events in said subject code; and

dividing a region of memory containing said subject code into at least one partition when identifying a self-modifying code event, wherein each partition includes a range of subject code addresses in said memory which are affected by a respective self-modifying code event.

19. The computer-readable storage medium of claim 18, wherein each partition further includes translated target

code corresponding to subject code contained in that partition.

20. The computer-readable storage medium of claim 18 or 19, wherein each said partition represents a region of memory that does not overlap with regions of memory corresponding to other partitions.

21. The computer-readable storage medium of claim 18, 19 or 20, wherein a self-modifying code event modifies a respective range of subject code addresses, said method further comprising:

modifying partitions existing in said memory that possess subject code addresses which are affected by said self-modifying code event.

22. The computer-readable storage medium of claim 21, wherein said partition modifying step comprises:

creating a new partition to include modified subject code corresponding to the self-modifying code event; and

for existing partitions having ranges of subject code addresses which overlap with the subject code addresses of the newly created partition, modifying said existing partitions to delete the subject code addresses from said existing partitions that overlap with the subject code addresses of the newly created partition such that the partitions no longer overlap.

23. The computer-readable storage medium of claim 22, wherein each partition further includes translated target

code corresponding to subject code contained in that partition, said method further comprising:

deleting translated target code associated with partitions that have been modified in response to the self-modifying code event; and

translating new target code for the subject code contained in the modified partitions.

24. The computer-readable storage medium of claim 22 or 23, said method further comprising adding translated target code to a partition as corresponding subject code in that partition is translated.

25. The computer-readable storage medium of any of claims 22 to 24, wherein each partition includes a particular range of subject code addresses that have been translated, such that the particular range of subject code addresses having been translated comprises an active range within the partition, said method further comprising:

determining whether the subject code addresses of said newly created partition overlap with any subject code addresses in said active range of any partition; and

for existing partitions having an active range that overlaps with the subject code addresses of said newly created partition, deleting translated target code associated with partitions that have been modified in response to the self-modifying code event, and translating new target code for the subject code contained in the modified partitions.

26. The computer-readable storage medium of claim 25, wherein each partition includes a range of subject code addresses that have not been translated referred to as an inactive range within the partition, said method further comprising:

for existing partitions having an active range which does not overlap with the subject code addresses of said newly created partition but having an inactive range that does overlap with the subject code addresses of said newly created partition, modifying said existing partitions to delete the subject code addresses from said inactive ranges in said existing partitions that overlap with the subject code addresses of the newly created partition such that the partitions no longer overlap, and leaving the translated target code associated with active ranges in said existing partitions unchanged.

27. The computer-readable storage medium of any of claims 21 to 26, further comprising:

identifying partitions that are adjacent to one another in memory having characteristics that allow them to be combined; and

aggregating said adjacent partitions into a single, combined partition.

28. The computer-readable storage medium of any of claims 18 to 27, wherein said self-modifying code event is identified during decoding of the subject code, said method further comprising inserting a special translation

structure into a control flow of the translated target code as a representation of the identified self-modifying code event.

29. The computer-readable storage medium of claim 28, in response to encountering said special translation structure during execution of the translated target code, said method further comprising:

identifying the range of subject code addresses affected by the self-modifying code event, and creating the partition in memory using this identified range of subject code addresses.

30. The computer-readable storage medium of any of claims 18 to 29, wherein each partition includes a pair of border guards which facilitate control flow passing between partitions, wherein said pair of border guards includes an entry border guard and an exit border guard, such that each exit border guard contains a specific reference to a counterpart entry border guard in a succeeding partition to be executed next.

31. The computer-readable storage medium of claim 30, when encountering an exit border guard during execution of a current partition, said method further comprising verifying that a counterpart entry border guard exists in a successive partition before passing control from the current partition to the successive partition.

32. The computer-readable storage medium of claim 30 or 31, wherein a set of border guards exists containing entry border guards and exit border guards for all

partitions, said method further comprising modifying said set of border guards whenever a new partition is created in response a self-modifying code event.

33. The computer-readable storage medium of any of claims 21 to 32, wherein when subject code defines a multi-threaded program, said method further comprising preventing other threads from entering a partition while the partition is being modified by another thread.

34. The computer-readable storage medium of any of claims 21 to 33 wherein each partition further includes translated target code corresponding to subject code contained in that partition, wherein each partition includes a pair of border guards which facilitate control flow passing between partitions, wherein said pair of border guards includes an entry border guard and an exit border guard, such that each exit border guard contains a specific reference to a counterpart entry border guard in a succeeding partition to be executed next, said method further comprising:

providing a memory management subsystem having regions which mirror the subject code partitions, wherein said memory management subsystem stores target code and border guard pairs associated with a partition along with its corresponding subject code; and

deleting an entire region of said memory management subsystem that corresponds to a specific partition whenever that specific partition is modified.

35. A translator apparatus for use in a target computing environment having a processor and a memory coupled to the processor for partitioning code during a translation of subject code into translated target code to account for self-modifying subject code, comprising:

a self-modifying code identifying mechanism configured for identifying self-modifying code events in said subject code; and

a partitioning mechanism configured for dividing a region of memory containing said subject code into at least one partition when identifying a self-modifying code event, wherein each partition includes a range of subject code addresses in said memory which are affected by a respective self-modifying code event.

36. The translator apparatus of claim 35, wherein each partition further includes translated target code corresponding to subject code contained in that partition.

37. The translator apparatus of claim 36, wherein each said partition represents a region of memory that does not overlap with regions of memory corresponding to other partitions.

38. The translator apparatus of claim 35, 36 or 37, wherein a self-modifying code event modifies a respective range of subject code addresses, said translator apparatus further comprising:

a partition modifying mechanism configured for modifying partitions existing in said memory that possess

subject code addresses which are affected by said self-modifying code event.

39. The translator apparatus of claim 38, wherein said partition modifying mechanism further:

creating a new partition to include modified subject code corresponding to the self-modifying code event; and

for existing partitions having ranges of subject code addresses which overlap with the subject code addresses of the newly created partition, modifying said existing partitions to delete the subject code addresses from said existing partitions that overlap with the subject code addresses of the newly created partition such that the partitions no longer overlap.

40. The translator apparatus of claim 39, wherein each partition further includes translated target code corresponding to subject code contained in that partition, said translator apparatus further comprising:

a target code deletion mechanism configured for deleting translated target code associated with partitions that have been modified in response to the self-modifying code event; and

a target code retranslation mechanism configured for translating new target code for the subject code contained in the modified partitions.

41. The translator apparatus of claim 39 or 40, further comprising a target code addition mechanism for

adding translated target code to a partition as corresponding subject code in that partition is translated.

42. The translator apparatus of claim 41, wherein each partition includes a particular range of subject code addresses that have been translated, such that the particular range of subject code addresses having been translated comprises an active range within the partition, said translator apparatus further comprising:

an active range detection mechanism configured for determining whether the subject code addresses of said newly created partition overlap with any subject code addresses in said active range of any partition; and

for existing partitions having an active range that overlaps with the subject code addresses of said newly created partition, said partition modifying mechanism further configured for deleting translated target code associated with partitions that have been modified in response to the self-modifying code event, and translating new target code for the subject code contained in the modified partitions.

43. The translator apparatus of claim 42, wherein each partition includes a range of subject code addresses that have not been translated referred to as an inactive range within the partition, said partition modifying mechanism further configured for:

for existing partitions having an active range which does not overlap with the subject code addresses of said

newly created partition but having an inactive range that does overlap with the subject code addresses of said newly created partition, modifying said existing partitions to delete the subject code addresses from said inactive ranges in said existing partitions that overlap with the subject code addresses of the newly created partition such that the partitions no longer overlap, and leaving the translated target code associated with active ranges in said existing partitions unchanged.

44. The translator apparatus of any of claims 38 to 43, further comprising:

an adjacent partition identifying mechanism configured for identifying partitions that are adjacent to one another in memory having characteristics that allow them to be combined; and

a partition aggregation mechanism configured for aggregating said adjacent partitions into a single, combined partition.

45. The translator apparatus of any of claims 35 to 44, wherein said self-modifying code event is identified during decoding of the subject code, said translator apparatus further comprising a notifying mechanism for inserting a special translation structure into a control flow of the translated target code as a representation of the identified self-modifying code event.

46. The translator apparatus of claim 45, in response to encountering said special translation structure during

execution of the translated target code, said translator apparatus configured for:

identifying the range of subject code addresses affected by the self-modifying code event, and creating the partition in memory using this identified range of subject code addresses.

47. The translator apparatus of claim 45 or 46, wherein each partition includes a pair of border guards which facilitate control flow passing between partitions, wherein said pair of border guards includes an entry border guard and an exit border guard, such that each exit border guard contains a specific reference to a counterpart entry border guard in a succeeding partition to be executed next.

48. The translator apparatus of claim 47, when encountering an exit border guard during execution of a current partition, said translator apparatus further configured for verifying that a counterpart entry border guard exists in a successive partition before passing control from the current partition to the successive partition.

49. The translator apparatus of claim 47 or 48, wherein a set of border guards exists containing entry border guards and exit border guards for all partitions, said translator apparatus further configured for modifying said set of border guards whenever a new partition is created in response a self-modifying code event.

50. The translator apparatus of any of claims 38 to 49, wherein when subject code defines a multi-threaded program, said translator apparatus further configured for preventing other threads from entering a partition while the partition is being modified by another thread.

51. The translator apparatus of any of claims 38 to 50, wherein each partition further includes translated target code corresponding to subject code contained in that partition, wherein each partition includes a pair of border guards which facilitate control flow passing between partitions, wherein said pair of border guards includes an entry border guard and an exit border guard, such that each exit border guard contains a specific reference to a counterpart entry border guard in a succeeding partition to be executed next, said translator apparatus further configured for:

providing a memory management subsystem having regions which mirror the subject code partitions, wherein said memory management subsystem stores target code and border guard pairs associated with a partition along with its corresponding subject code; and

deleting an entire region of said memory management subsystem that corresponds to a specific partition whenever that specific partition is modified.

ABSTRACT

METHOD AND APPARATUS FOR PARTITIONING CODE IN PROGRAM CODE CONVERSION

A partitioning technique utilized by a translator to divide the subject code space into regions, referred to as partitions, where each partition contains a distinct set of basic blocks of subject code and corresponding target code. The partitioning technique divides the translator's representation of subject code and subject code translations into non-overlapping regions of subject memory. In this manner, when the subject program modifies subject code, only those partitions actually affected by the self-modifying code need be discarded and all translations in unaffected partitions can be kept. This partitioning technique is advantageous in limiting the amount of target code that must be retranslated in response to self-modifying code operation. In another process, the partitioning technique allows multithreaded subject programs that also involve self-modifying code to perform code modification in a thread-safe manner.

[Figure 11]

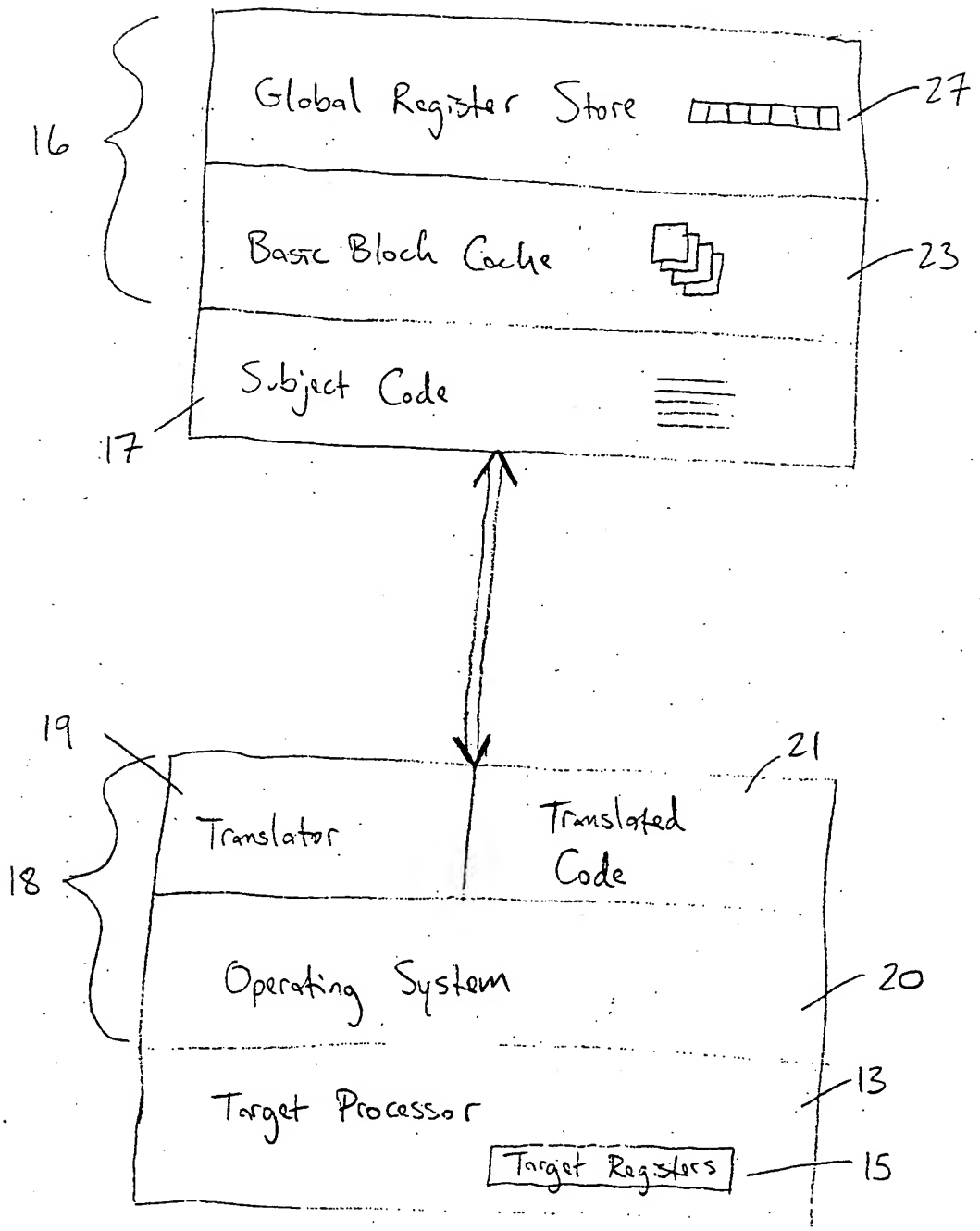

Global Register Store
27
16
Basic Block Cache
23
Subject Code
17
19
21
Translator
Translated Code
18
Operating System
20
Target Processor
13
Target Registers
15

FIG. 1

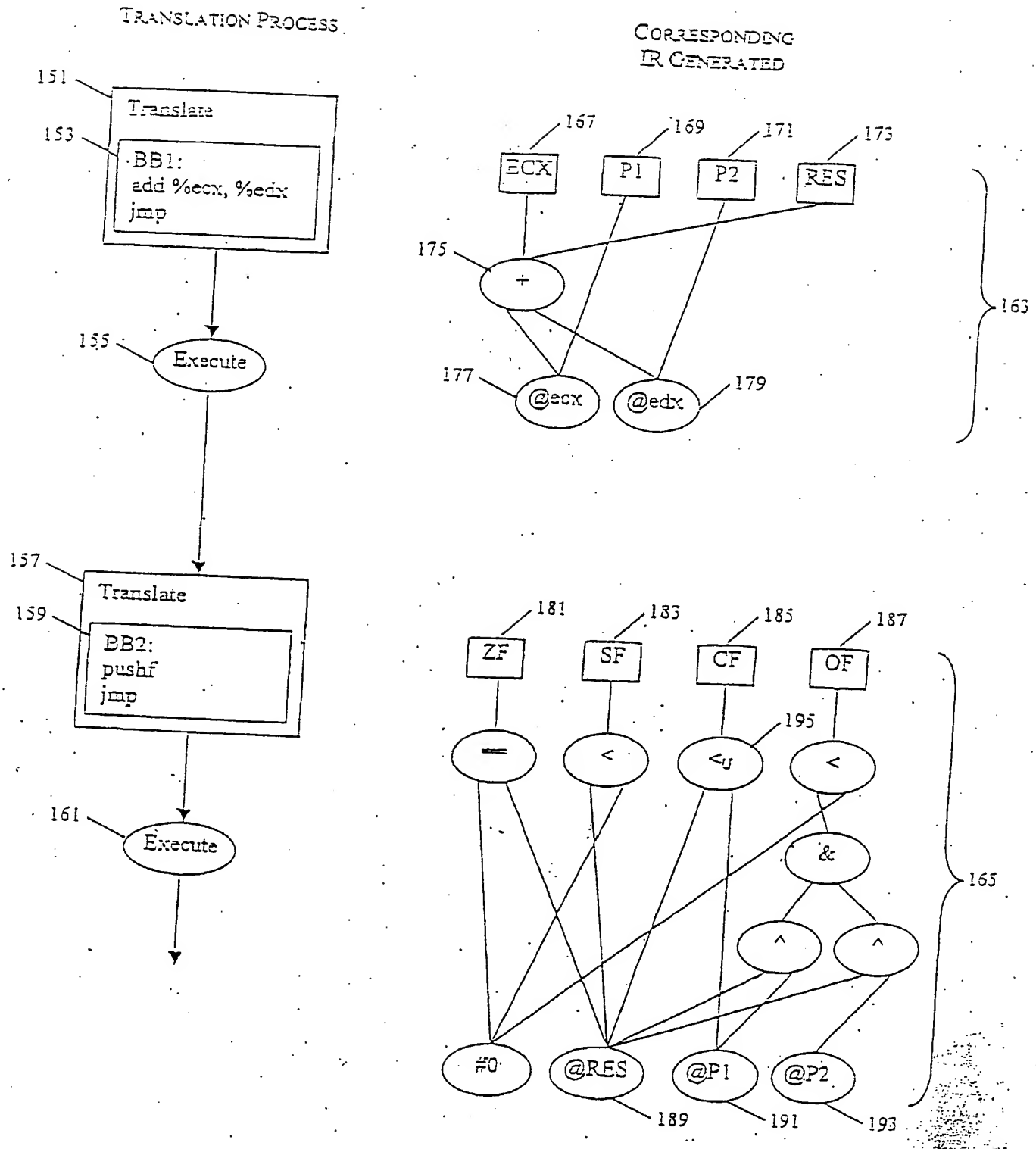
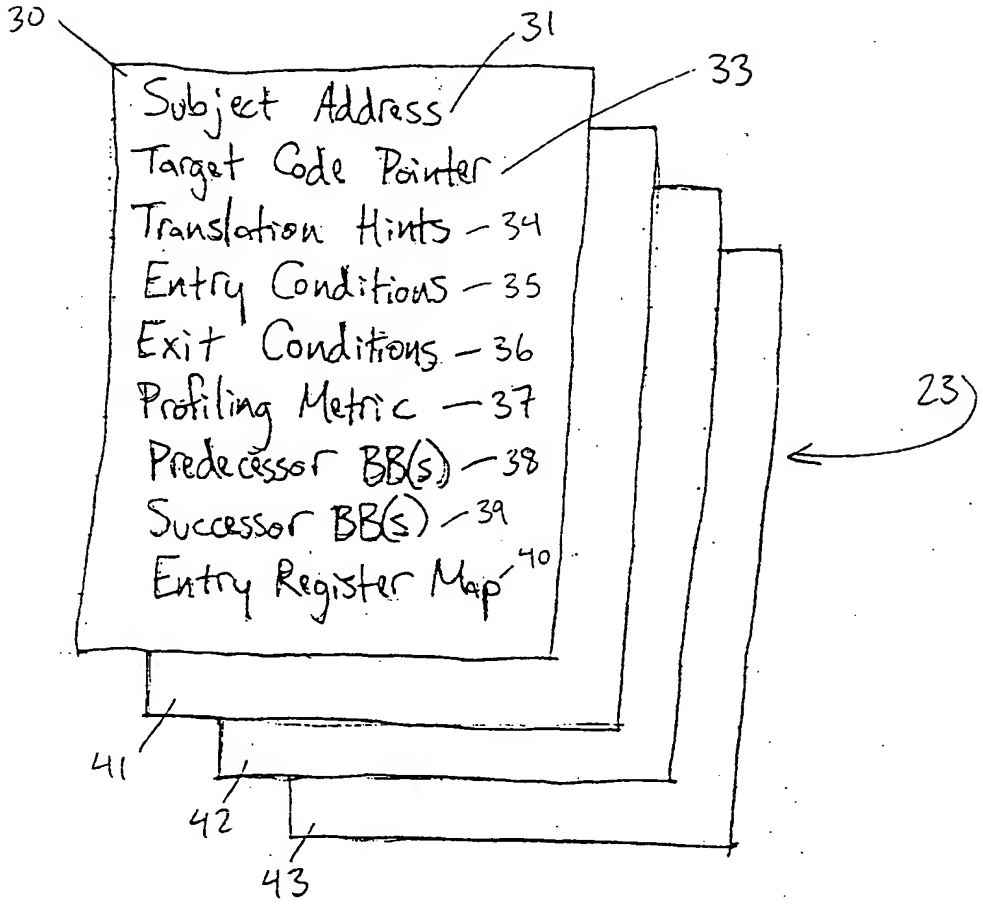


FIG. 2



BASIC BLOCK DATA STRUCTURE &
BASIC BLOCK CACHE

FIG. 3

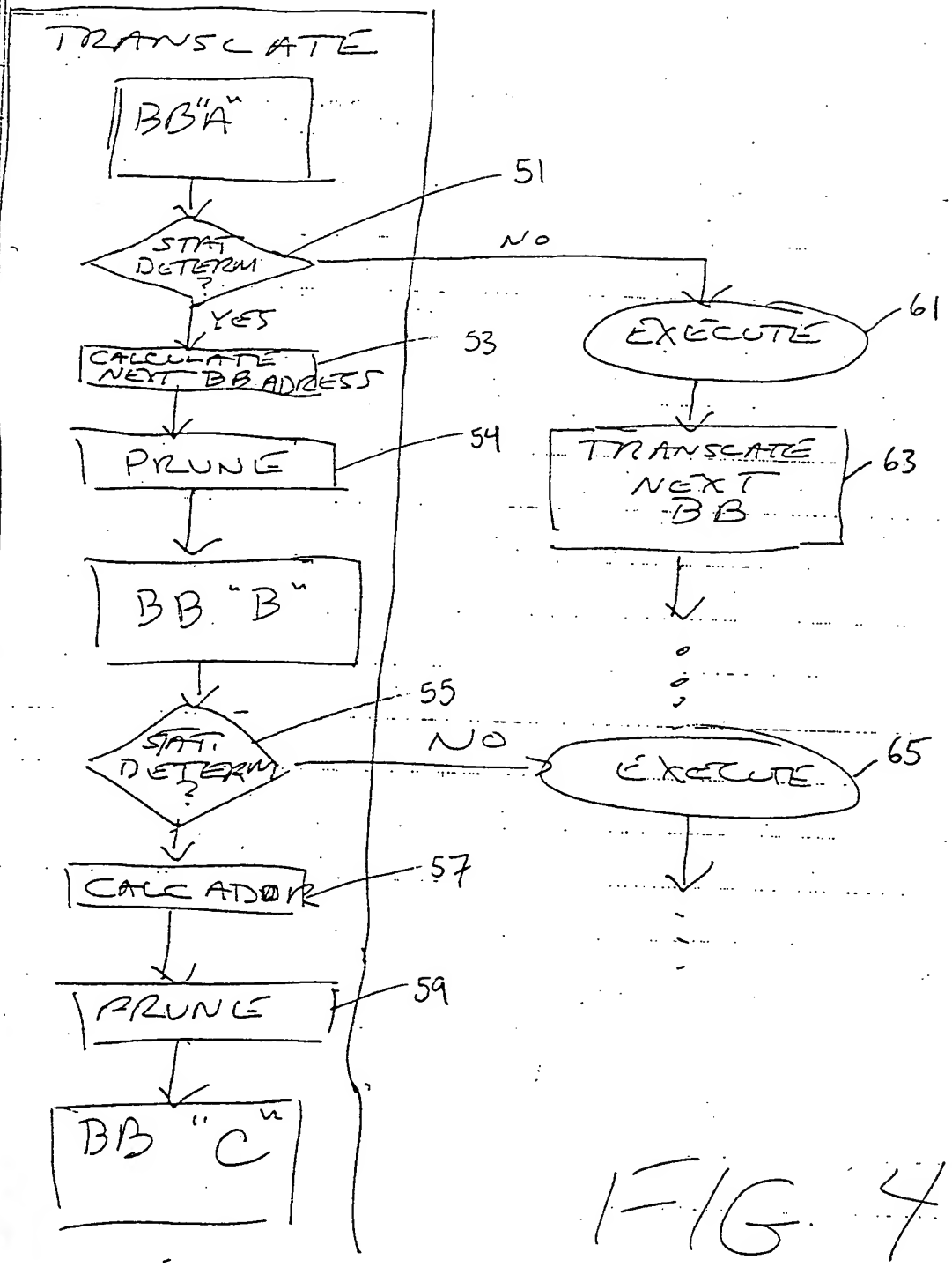


FIG 4

EXTENDED BASIC BLOCKS

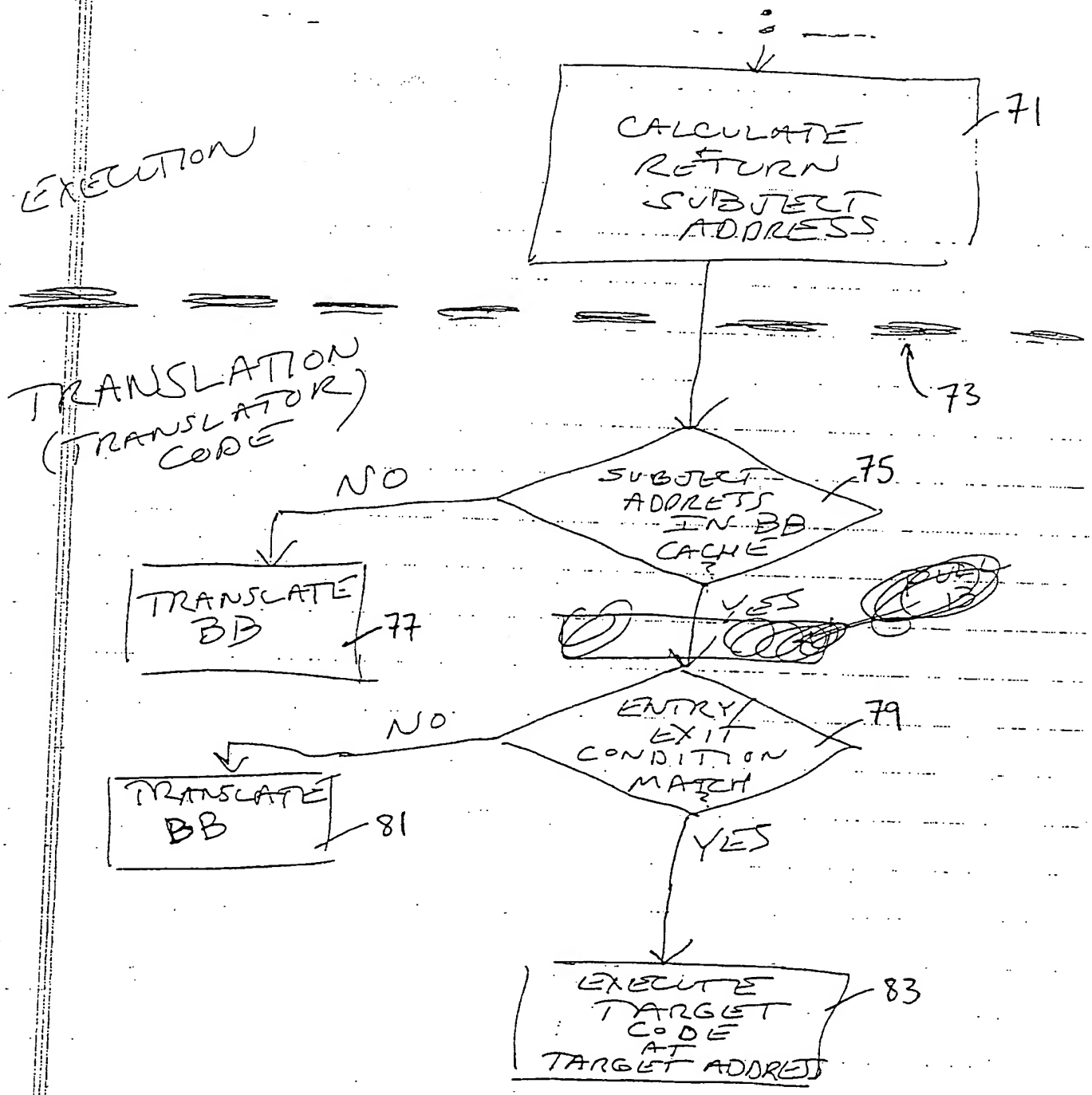
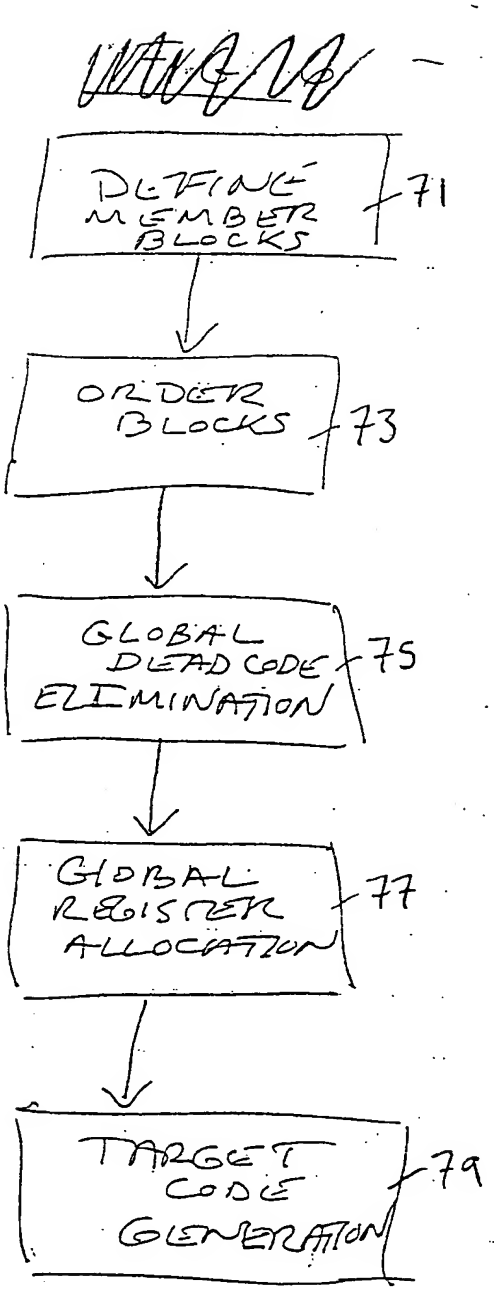


FIG 5

ISOBLOCKING



GROUP BLOCKS

FIG. 6

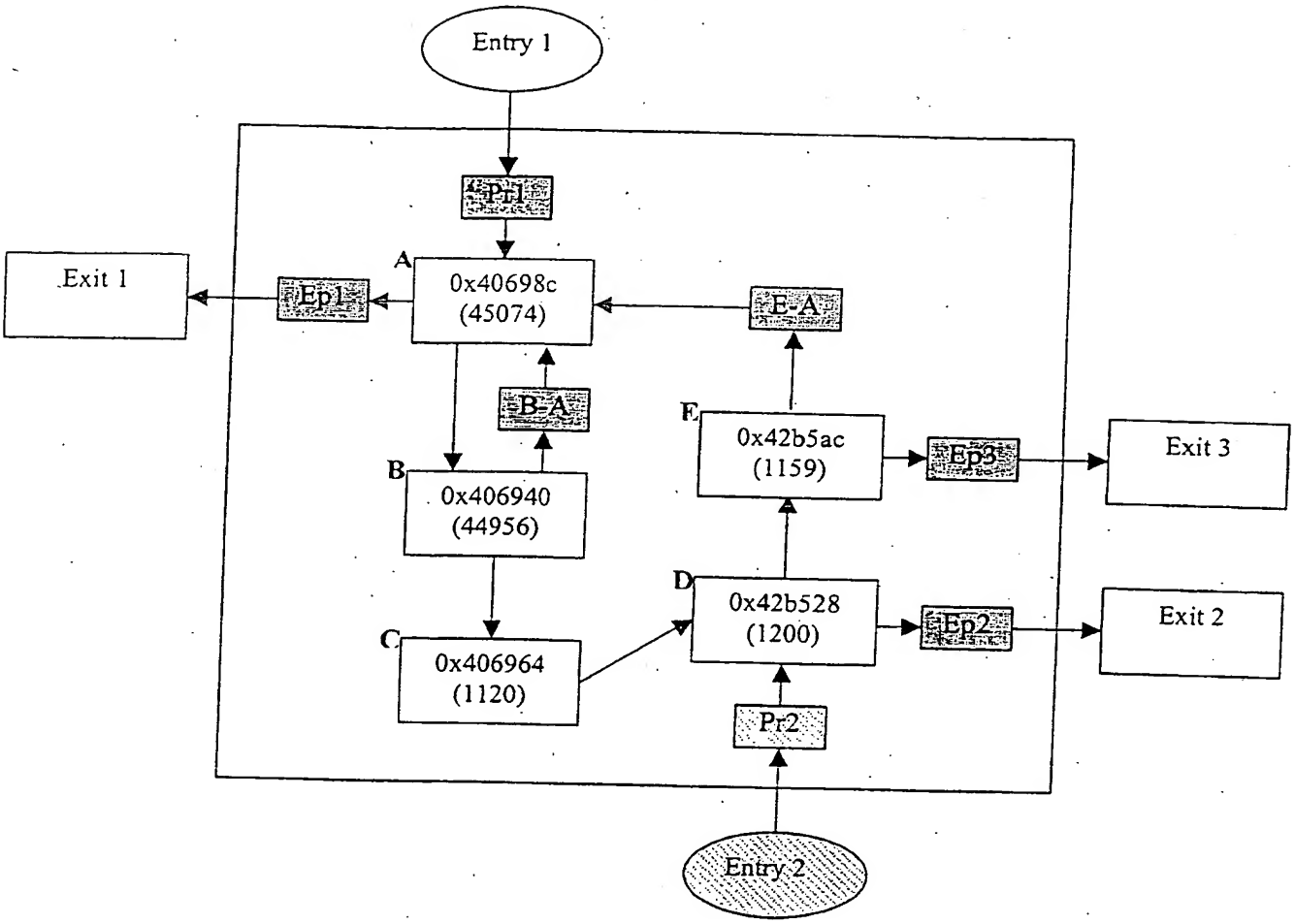

Entry 1
Pr1
A
0x40698c
(45074)
Exit 1
Ep1
E-A
B-A
B
0x406940
(44956)
F
0x42b5ac
(1159)
Ep3
Exit 3
D
0x42b528
(1200)
Ep2
Exit 2
C
0x406964
(1120)
Pr2
Entry 2

FIG. 7

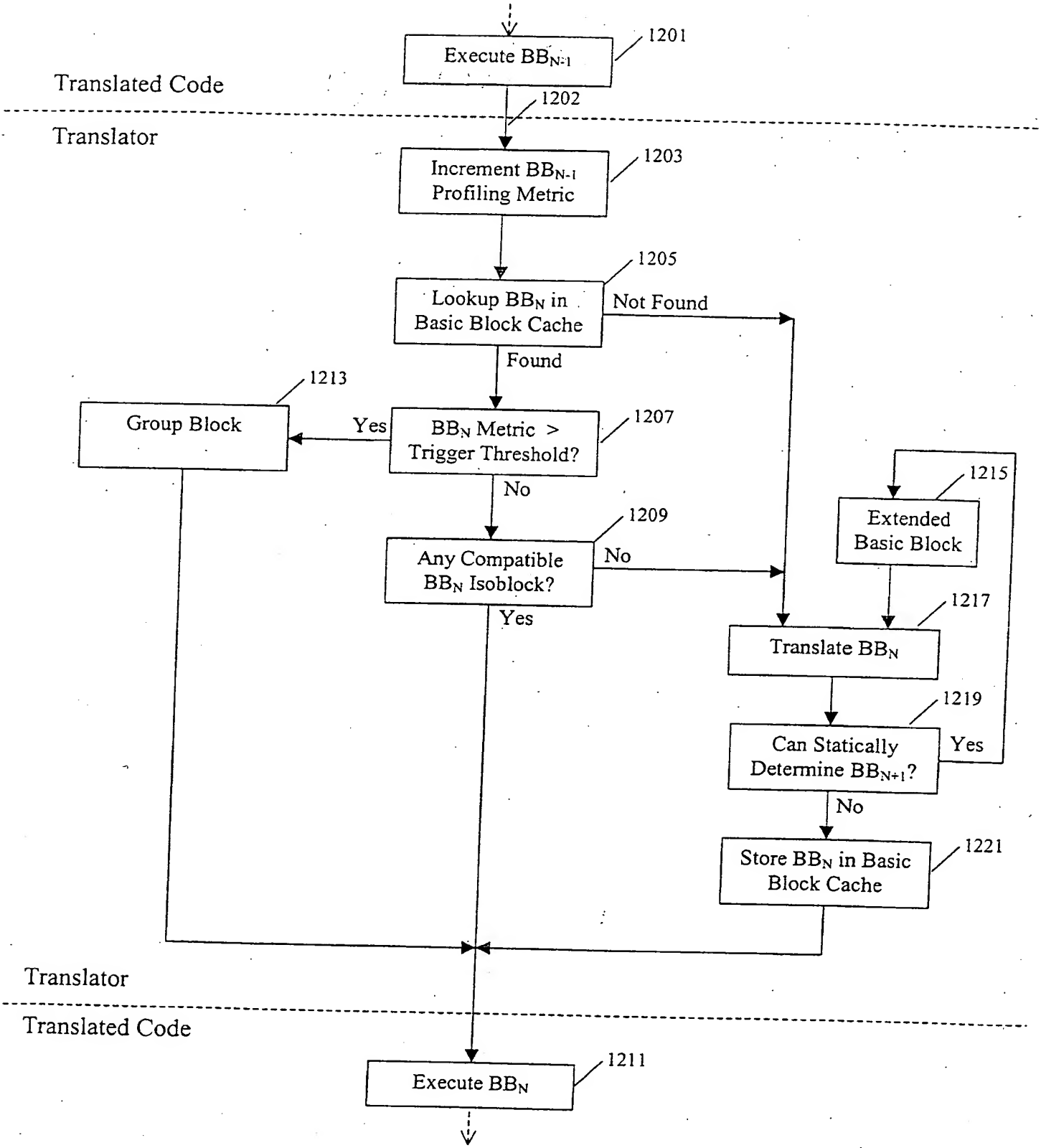

Translated Code
Translator
Execute BB_{N-1}
1201
1202
Increment BB_{N-1} Profiling Metric
1203
Lookup BB_N in Basic Block Cache
1205
Not Found
Found
BB_N Metric > Trigger Threshold?
1207
Yes
Group Block
1213
No
Any Compatible BB_N Isoblock?
1209
No
Yes
Extended Basic Block
1215
Translate BB_N
1217
Can Statically Determine BB_{N+1}?
1219
Yes
No
Store BB_N in Basic Block Cache
1221
Translator
Translated Code
Execute BB_N
1211

FIG. 8

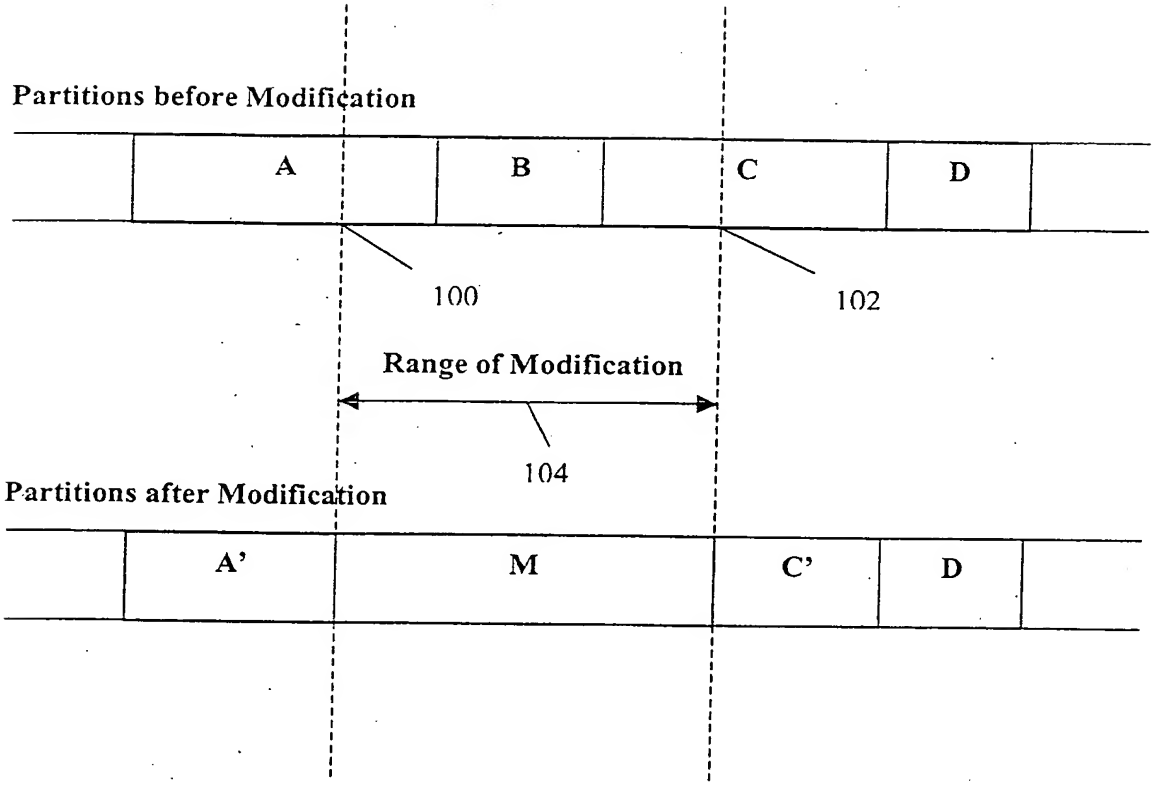

Partitions before Modification
A
B
C
D
100
102
Range of Modification
104
Partitions after Modification
A'
M
C'
D

FIG. 9

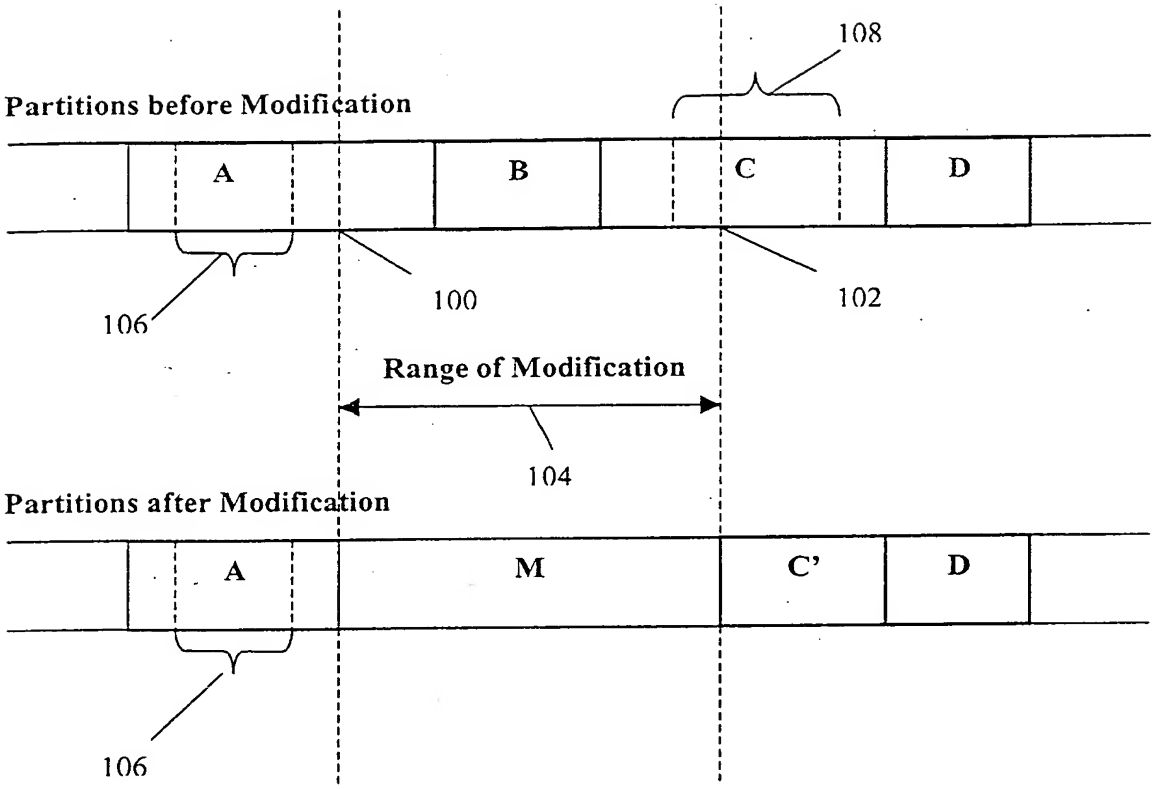
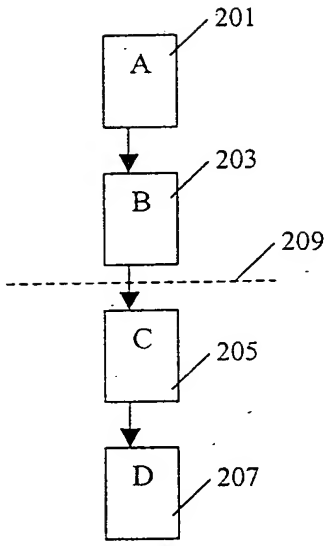
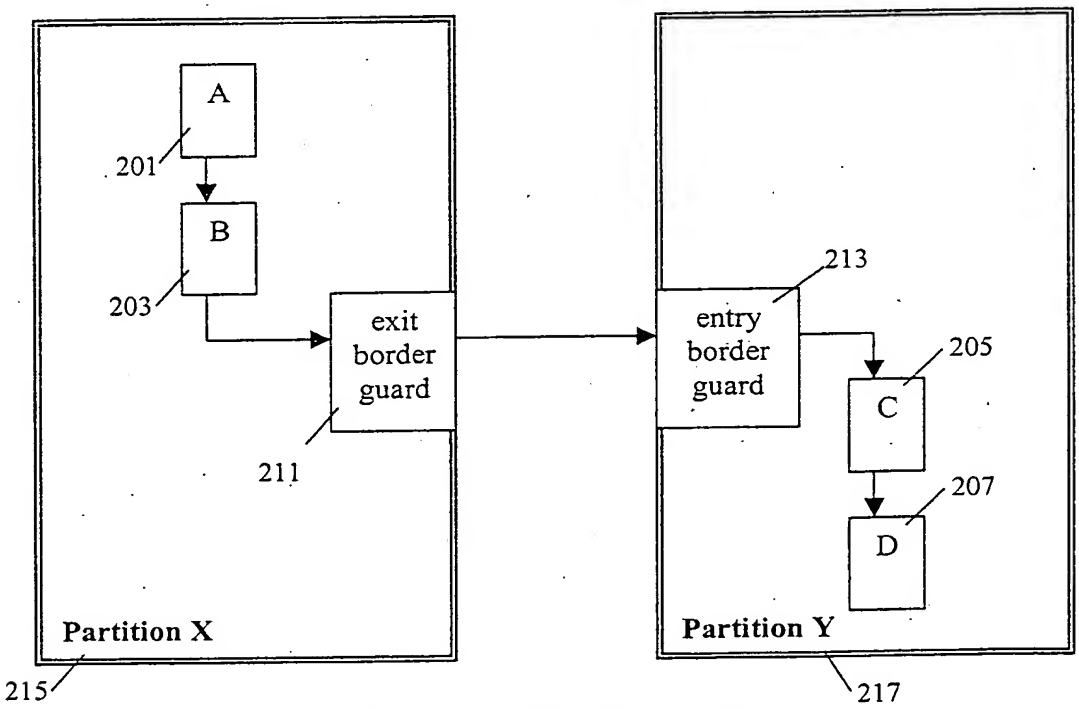

Partitions before Modification
A
B
C
D
108
106
100
102
Range of Modification
104
Partitions after Modification
A
M
C'
D
106

FIG. 10



TRANSLATOR WITHOUT PARTITIONS



TRANSLATOR WITH PARTITIONS

FIG. 11